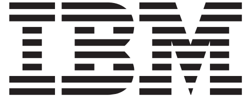
WebSphere Application Server for z/OS V5.0.2:



Applications

SA22-7959-01

Note

Before using this information, be sure to read the general information under "Notices" on page 925.

Compilation date: December 3, 2003

© Copyright International Business Machines Corporation 2003. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

How to send your comments xi

Chapter 1. Welcome to Applications . . 1

Chapter 2. Using Web applications . . . 7
Web applications 7
web.xml file 8
Migrating Web application components 9
Default Application. 12
Servlets. 13
Developing servlets with WebSphere Application Server extensions 13
Application lifecycle listeners and events . . . 14
Listener classes for servlet context and session changes. 14
Example: com.ibm.websphere.DBConnectionListener.java . 15
Servlet filtering 15
Filter, FilterChain, FilterConfig classes for servlet filtering. 16
Example: com.ibm.websphere.LoggingFilter.java 16
Configuring page list servlet client configurations 17
autoRequestEncoding and autoResponseEncoding 20
Examples: autoRequestEncoding and autoResponseEncoding encoding examples . . . 21
JavaServer Pages files 21
Developing JavaServer Pages files with WebSphere extensions 22
Tag libraries 22
tsx:dbconnect tag JavaServer Pages syntax . . . 22
dbquery tag JavaServer Pages syntax 24
dbmodify tag JavaServer Pages syntax 25
tsx:getProperty tag JavaServer Pages syntax and examples 25
tsx:userid and tsx:passwd tag JavaServer Pages syntax 26
tsx:repeat tag JavaServer Pages syntax 27
Example: Combining tsx:repeat and tsx:getProperty JavaServer Pages tags. 27
Example: tsx:dbmodify tag syntax 28
Example: Using tsx:repeat JavaServer Pages tag to iterate over a results set 28
JspBatchCompiler tool 30
Bean Scripting Framework 31
Developing Web applications 32
Disabling JavaServer Pages run-time compilation 32
Example: Converting JavaScript source to the Bean Scripting Framework 34
Scenario: Creating a Bean Scripting Framework application 35
Example: Bean Scripting Framework code example 40
Web modules. 43
Assembling Web applications 44
Using the AAT to assemble Web modules 44
Context parameters. 46
Security constraints. 46
Servlet mappings 47
Invoker attributes 47
Error pages 47
File serving 47
Initialization parameters 47
Servlet caching 47
Web components 47
Web property extensions 47
Web resource collections 47
Welcome files. 48
Context parameter assembly settings 48
Initialization parameter assembly settings . . . 48
Filter assembly settings 49
JavaServer Pages attribute assembly settings . . 49
Multipurpose Internet Mail Extensions (MIME) filter assembly settings 52
Page list assembly settings 53
Security constraint assembly settings 54
Servlet mapping assembly settings 55
Tag library assembly settings 55
Welcome file assembly settings 55
Servlet caching configuration assembly settings 56
Web components assembly settings 58
Web modules assembly settings 60
Assembly property extensions 63
File serving attribute assembly settings 63
Invoker attribute assembly settings 63
Error page assembly settings 64
Web resource collections security constraint properties 64
Troubleshooting tips for Web application deployment 65
Modifying the default Web container configuration 66
Web container 67
Web container settings. 67
Web module settings 68
Web Module Deployment settings 68
Web container advanced settings 69
Web container custom properties 70
Web applications: Resources for learning 71

Chapter 3. Managing HTTP sessions 73
Sessions 73
Migrating HTTP sessions 74
Developing session management in servlets . . . 75
Example: SessionSample.java 76
Assembling so that session data can be shared . . 76
Session security support 77
Session management support 78
Configuring session management by level 80
Session tracking options 80
Session tracking with cookies 80
Session tracking with URL rewriting 80
Session tracking with SSL information 82

© Copyright IBM Corp. 2003

Configuring session tracking 82
Serializing access to session data 82
Session Management settings 83
Cookie settings 86
Distributed sessions 87
Session recovery support 87
Distributed Environment settings 88
Configuring for database session persistence . . . 88
Switching to a multirow schema 89
Creating a DB2 table for session persistence . . 89
Database settings 91
Multirow schema considerations 92
Memory-to-memory replication. 93
Clustered session support 94
Tuning session management 95
Configuring scheduled invalidation 95
Configuring write contents 96
Configuring write frequency. 97
Base in-memory session pool size 97
Controlling write operations. 98
Tuning parameter settings 98
Tuning parameter custom settings 99
Best practices for using HTTP Sessions 101
Managing HTTP sessions: Resources for learning: 104

Chapter 4. Using enterprise beans in applications 105

Enterprise beans 105
Developing enterprise beans 106
Migrating enterprise bean code to the supported specification 107
WebSphere extensions to the Enterprise JavaBeans specification 110
Best practices for developing enterprise beans 110
Unknown primary-key class 115
Using access intent policies 115
Access intent policies 115
Applying access intent policies to methods . . 118
Access intent exceptions 118
Access intent assembly settings 119
Access intent best practices 121
Frequently asked questions: Access intent . . . 122
EJB modules. 123
Assembling EJB modules 123
Assembling EJB modules 124
Container transactions 126
Method extensions 126
Method permissions 126
References 126
CMP field assembly settings 126
Container transaction assembly settings . . . 126
EJB module assembly settings 128
Entity bean assembly settings 130
Message-driven bean assembly settings. . . . 138
Method extension assembly settings 140
Method permission assembly settings 143
Query assembly settings. 144
EJB reference assembly settings 144
EJB local-reference assembly settings 145
EJB relation assembly settings 146
Exclude list assembly settings 146
Security role assembly settings 147
Session bean assembly properties. 148
EJB containers 151
Managing EJB containers 152
EJB container settings 153
EJB container system properties 154
EJB cache settings 154
Container interoperability 155
Deploying EJB modules 159
EJB module collection 159
EJB module settings 159
Troubleshooting tips for EJBDEPLOY relationships. 159
Enterprise beans: Resources for learning 160
EJB method Invocation Queuing 161

Chapter 5. Using message-driven beans in applications 163

Message-driven beans - an overview 163
Message-driven beans - components. 164
Message-driven beans - transaction support . . 166
Designing an enterprise application to use message-driven beans 166
Developing an enterprise application to use message-driven beans 168
Deploying an enterprise application to use message-driven beans 170
Configuring deployment attributes using the Assembly Toolkit 170
Configuring deployment attributes for a message-driven bean 173
Configuring message listener resources for message-driven beans 175
Configuring the message listener service . . . 175
Adding a new listener port. 182
Configuring a listener port 183
Deleting a listener port 183
Configuring security for message-driven beans 183
Administering listener ports 184
Important files for message-driven beans and extended messaging 186
Troubleshooting message-driven beans 186
Message-driven beans samples 187

Chapter 6. Using application clients 189

Application clients 189
Application client functions 191
J2EE application clients 192
Pluggable application clients 193
Migration tips for application clients 195
Developing J2EE application client code 195
J2EE application client class loading 198
Developing pluggable application client code. . . 200
Assembling application clients. 201
Assembling Application Client Modules 202
Application client assembly settings 203
Deploying application clients on z/OS 203
Application Client Resource Configuration Scripting tool for z/OS 205

Determining required properties for z/OS application client resources 207
Deploying application clients on workstation platforms 215
Starting the Application Client Resource Configuration Tool and opening an EAR file . . 215
Data sources for application clients 216
Configuring new data source providers (JDBC providers) for application clients 216
Configuring new data sources for application clients 219
Configuring mail providers and sessions for application clients 220
Configuring new mail sessions for application clients 223
URLs for application clients 223
URL providers for the Application Client Resource Configuration Tool 224
Configuring new URL providers for application clients 224
Configuring new URLs with the Application Client Resource Configuration Tool 226
WebSphere asynchronous messaging using the Java Message Service API for the Application Client Resource Configuration Tool 227
Configuring Java messaging client resources . . 227
Configuring new connection factories for application clients 262
Configuring new Java Message Service destinations for application clients 263
Example: Configuring MQ Queue and Topic connection factories and destination factories for application clients 263
Example: Configuring WAS Queue and Topic connection factories and destination factories for application clients 265
Configuring new resource environment providers for application clients 266
Configuring new resource environment entries for application clients. 267
Managing application clients 268
Updating data source and data source provider configurations with the Application Client Resource Configuration Tool 268
Updating URLs and URL provider configurations for application clients 269
Updating mail session configurations for application clients 269
Updating Jave Message Service provider, connection factories, and destination configurations for application clients 270
Updating MQ Java Message Service provider, MQ connection factories, and MQ destination configurations for application clients 270
Updating Resource Environment Entry and Resource Environment Provider configurations for application clients. 271
Removing application client resources 272
Running application clients. 272
launchClient tool 273
Application client troubleshooting tips 277

Chapter 7. Using Web services based on Web Services for J2EE 283

Web services 284
SOAP 285
Planning to use Web services based on Web Services for J2EE 286
Service-oriented architecture 287
Web services approach to a service-oriented architecture 287
Web services business models supported . . . 290
Migrating Apache SOAP Web services to Web Services for J2EE 291
Developing Web services based on Web Services for J2EE 293
Example: Developing Web services based on Web Services for J2EE 294
Web Services for J2EE 296
Java API for XML-based remote procedure call (JAX-RPC) 297
Artifacts used to develop Web services based on Web Services for J2EE 297
Mapping between Java language, WSDL and XML 298
Installing IBM Web Services Development Kit for z/OS 321
Java2WSDL command 321
WSDL2Java command 324
Setting up a development and unmanaged client execution environment for Web services based on Web Services for J2EE 327
Developing a Web service from a Java bean . . 328
Developing a Web service using a stateless session enterprise bean 339
Configuring the webservices.xml deployment descriptor 342
Configuring the ibm-webservices-bnd.xmi deployment descriptor 343
Configuring the webservices.xml deployment descriptor for Handler classes 345
Developing a new Web service with an existing WSDL file using a Java bean 346
Developing a new Web service from an existing WSDL file using a stateless session enterprise bean 348
Web services implementation scope 349
Default Port Mapping Definitions collection . . 350
Default Port Type Mapping Properties settings 350
Developing Web services clients based on Web Services for J2EE 351
Example: Developing Web services clients based on Web Services for J2EE 351
Developing client bindings from a WSDL file 352
Assembling a Web services-enabled client JAR file into an EAR file 353
Assembling a Web services-enabled client WAR file into an EAR file 355
Configuring the ibm-webservicesclient-bnd.xmi deployment descriptor 356
Configuring the webservicesclient.xml deployment descriptor 359

Configuring the webservicesclient.xml deployment descriptor for Handler classes . . . 360
Testing Web services-enabled clients 364
Web services client bindings 365
Assembling Web services applications based on Web Services for J2EE 366
Assembling a Web services-enabled EJB JAR file 366
Assembling a Web services-enabled WAR file 368
Assembling a Web services-enabled EJB JAR into an EAR file 371
Assembling a Web services-enabled WAR into an EAR file 371
Enabling a Web services-enabled EAR file . . . 372
Deploying Web services based on Web Services for J2EE 377
wsdeploy command 378
Using the Java Messaging Service to transport Web services requests 380
Java Messaging Service endpoint URL syntax 382
Securing Web services based on WS-Security . . . 383
Web services security specification- a chronology 384
Web services security support 385
Web services security and Java 2 Platform, Enterprise Edition security relationship 387
Web services security model in WebSphere Application Server 390
Web services security property collection . . . 393
Web services security property configuration settings 393
Usage scenario for propagating security tokens 394
Configurations 395
Authentication method overview 411
XML digital signature 414
Securing Web services using XML digital signature 419
XML encryption 488
Securing Web services using XML encryption 491
Securing Web services using basicauth authentication 512
Identity assertion 520
Securing Web services using identity assertion authentication 521
Securing Web services using signature authentication 528
Token type overview 534
Security token 538
Securing Web services using a pluggable token 538
Tuning Web services based on Web Services for J2EE 550
Troubleshooting Web services based on Web Services for J2EE 551
Troubleshooting command-line tools for Web services based on Web Services for J2EE . . . 551
Troubleshooting compiled bindings for Web services based on Web Services for J2EE . . . 552
Troubleshooting the run time of Web services based on Web Services for J2EE 552
Troubleshooting the run time for a Web services client based on Web Services for J2EE 554
Troubleshooting serialization and deserializaton in Web services based on Web Services for J2EE . 554
Frequently asked questions about Web services based on Web Services for J2EE 556
Web services: Resources for learning. 558

Chapter 8. Web Services Invocation Framework (WSIF): Enabling Web services. 563
Goals of WSIF 563
WSIF - Web services are more than just SOAP services 564
WSIF - Tying client code to a particular protocol implementation is restricting 564
WSIF - Incorporating new bindings into client code is hard 564
WSIF - Multiple bindings can be used in flexible ways 564
WSIF - Enabling a freer Web services environment promotes intermediaries 565
An overview of WSIF 565
WSIF architecture 566
Using WSIF with Web services that offer multiple bindings 566
WSIF and WSDL 566
WSIF usage scenarios. 567
Dynamic invocation 568
Using WSIF to invoke Web services 568
Using the WSIF providers 569
Developing a WSIF service 580
Using complex types 589
Using the Java Naming and Directory Interface (JNDI) 591
Passing SOAP messages with attachments using WSIF 593
Interacting with the J2EE container in WebSphere Application Server. 595
Running WSIF as a client 595
WSIF system management and administration . . 596
Maintaining the WSIF properties file 596
Enabling security for WSIF 597
Troubleshooting the Web Services Invocation Framework 597
WSIF API 602
WSIF API reference: Creating a message for sending to a port 602
WSIF API reference: Finding a port factory or service. 603
WSIF API reference: Using ports 605

Chapter 9. IBM WebSphere UDDI Registry. 611
UDDI Registry terminology. 611
UDDI Registry definitions 612
An overview of IBM UDDI Registries 612
Installing and setting up a UDDI Registry 614
Installing the UDDI Registry into a deployment manager cell. 616
Installing the UDDI Registry into a single appserver 624

Reinstalling the UDDI Registry application 630
Removing the UDDI Registry application from a deployment manager cell 631
Removing the UDDI Registry application from a single application server. 631
Configuring the UDDI Registry 632
Configuring global UDDI properties. 632
Modifying the database userid and password 634
Configuring security roles 634
Configuring the UDDI User Console (GUI) for multiple language encoding support. 635
Customizing the UDDI User Console (GUI) . . 635
Configuring SOAP interface properties 636
Configuring SOAP properties with the Application Assembly ToolWebSphere Assembly Toolkit or the Application Assembly Tool . . . 636
Configuring SOAP properties in an application that is already deployed. 637
Administering the UDDI Registry 637
Running the UDDI Registry 637
Backing up and restoring the UDDI Registry database 638
UDDI user console 638
Displaying the user console 642
Custom Taxonomy Support in the UDDI Registry 643
SOAP application programming interface for the UDDI Registry 652
Programming the SOAP API 653
SOAP API error handling tips in the UDDI Registry 653
UDDI Registry Application Programming Interface 653
Inquiry API for the UDDI Registry 653
Publish API for the UDDI Registry 656
UDDI EJB Interface for the UDDI Registry . . . 657
Datatypes package in the UDDI Registry . . . 663
EJB interface methods in the UDDI Registry . . 666
UDDI troubleshooting tips 666
Turning on UDDI trace 670
Messages. 670
UDAI (Web Services UDDI) messages 671
UDCF (Web Services UDDI) messages 671
UDDA (Web Services UDDI) messages 672
UDDM (Web Services UDDI) messages. . . . 672
UDEJ (Web Services UDDI) messages 672
UDEX (Web Services UDDI) messages 673
UDIN (Web Services UDDI) messages 673
UDLC (Web Services UDDI) messages 696
UDPR (Web Services UDDI) messages 696
UDRS (Web Services UDDI) messages 696
UDSC (Web Services UDDI) messages 696
UDSP (Web Services UDDI) messages 696
UDUC (Web Services UDDI) messages 698
UDUT UDDI Utility Tools messages. 700
UDUU (Web Services UDDI) messages 712
Running the UDDI samples 712
Installation Verification Program (IVP) 712
Reporting problems with the IBM WebSphere UDDI Registry 714
Feedback 715

Chapter 10. Class loading 717
Class loaders 718
Class loader collection 721
Classloader ID 722
Classloader Mode 722
Class loader settings 722
Migrating the class-loader Module Visibility Mode setting. 722
Class loading: Resources for learning 723

Chapter 11. Using EJB query. 725
EJB query language 725
Example: EJB queries. 726
FROM clause 728
Inheritance in EJB query. 729
Path expressions 729
WHERE clause 730
Scalar functions 738
Aggregation functions 741
SELECT clause 742
ORDER BY clause 743
Subqueries 743
EJB query restrictions. 744
EJB Query: Reserved words 745
EJB query: BNF syntax 745
Comparison of EJB 2.0 specification and WebSphere query language. 747

Chapter 12. Internationalizing applications 749
Internationalization 749
Identifying localizable text 750
Creating message catalogs 750
Composing language-specific strings 751
Localization API support 751
LocalizableTextFormatter class. 752
Creating a formatter instance 755
Setting optional localization values 756
Generating localized text 758
Preparing the localizable-text package for deployment 758
LocalizableTextEJBDeploy command 759
Internationalization: Resources for learning . . . 760

Chapter 13. Using the transaction service 761
Transaction support in WebSphere Application Server 761
Resource manager local transaction (RMLT) . . 763
Global transactions 763
Local transaction containment (LTC). 764
Local and global transaction considerations . . 764
Developing components to use transactions . . . 765
Configuring transactional deployment attributes using the Assembly Toolkit. 765
Configuring transactional deployment attributes using the Application Assembly Tool 767
Using bean-managed transactions 768
Classifying WebSphere transaction workload for WLM 769

Configuring transaction properties for an application server 771
Transaction service settings 772
Using local transactions 774
Managing active transactions 775
Interoperating transactionally between application servers 776
Troubleshooting transactions 776
Transaction service exceptions 777
UserTransaction interface - methods available . . 778

Chapter 14. Using naming 779

Naming 780
Version 5 features for name space support. . . . 780
Name space logical view 781
Initial context support 784
Lookup names support in deployment descriptors and thin clients. 785
JNDI support in WebSphere Application Server 788
Developing applications that use JNDI 788
Example: Getting the default initial context . . 790
Example: Getting an initial context by setting the provider URL property 794
Example: Setting the provider URL property to select a different root context as the initial context 796
Example: Looking up an EJB home with JNDI 797
Example: Looking up a JavaMail session with JNDI 799
JNDI interoperability considerations. 799
JNDI caching 801
JNDI cache settings 802
Example: Controlling JNDI cache behavior from a program 803
JNDI name syntax. 804
INS name syntax 804
JNDI to CORBA name mapping considerations 805
Example: Setting the syntax used to parse name strings. 805
Developing applications that use CosNaming (CORBA Naming interface). 806
Example: Getting an initial context with CosNaming 806
Example: Looking up an EJB home with CosNaming 809
Configured name bindings 811
Name space federation 813
Name space bindings. 814
Configuring and viewing name space bindings . . 814
String binding settings 815
CORBA object binding settings 815
Indirect lookup binding settings 816
EJB binding settings 816
Name space binding collection 817
Configuring name servers 818
Name server settings 818
Troubleshooting name space problems 818
dumpNameSpace tool 819
Example: Invoking the name space dump utility 821
Name space dump utility for `java:`, `local:` and `server` name spaces 822
Example: Invoking the name space dump utility for `java:` and `local:` name spaces 824
Name space dump sample output 825
Naming and directories: Resources for learning . . 827

Chapter 15. Using the dynamic cache service to improve performance . . . 829

Dynamic cache 829
Configuring cache replication 829
Cache replication 830
Internal messaging configuration settings . . . 830
Enabling the dynamic cache service 831
Dynamic cache service settings 831
Configuring servlet caching 832
Configuring the dynamic cache disk offload . . 833
Configuring Edge Side Include caching. . . . 833
Configuring external cache groups 836
Displaying cache information 840
Configuring cacheable objects with the cachespec.xml file 841
Verifying the cacheable page 843
Cachespec.xml file. 843
Configuring command caching 850
Command class 850
CacheableCommandImpl class 851
Example: Caching a command object 851
Example: Caching Web services 852
Example: Configuring the dynamic cache 854
Cache monitor 856
Edge cache statistics 858
Troubleshooting the dynamic cache service . . . 858
Troubleshooting tips for the dynamic cache service. 859

Chapter 16. Assembling applications with the AAT. 863

Application assembly and J2EE applications . . . 864
Archive support in Version 5.0 865
Starting the Application Assembly Tool (AAT) . . 865
Migrating application modules from J2EE 1.2 to J2EE 1.3 866
earconvert tool 866
Assembling new or modifying existing modules 867
Adding files to assembled modules 869
Resource environment reference assembly settings 870
Resource Adapter Archive file assembly settings 871
Saving applications after assembly 874
Verifying archive files 874
Application assembly performance checklist . . . 875
Generating code for deployment 875
ejbdeploy tool 876
ejbdeploy syntax -- relationship to Application Assembly Tool options 877
Application Assembly Tool: Resources for learning 877

Chapter 17. Assembling applications with the Assembly Toolkit 879

Application assembly and J2EE applications . . . 881
Archive support in Version 5.0 882

Starting the Assembly Toolkit 882
astk command 883
Migrating code artifacts to the Assembly Toolkit 883
Importing enterprise applications. 884
Importing WAR files 884
Importing client applications 885
Importing EJB files 885
Importing RAR files or connectors 886
Creating enterprise applications 886
Creating Web applications 887
Creating application clients. 889
Creating EJB modules 890
Creating connector modules 891
Editing deployment descriptors 892
Mapping enterprise beans to database tables . . . 893
Verifying archive files 894
Generating code for EJB deployment 895
Generating code for Web service deployment. . . 896
Assembly Toolkit: Resources for learning 896

Chapter 18. Deploying and managing applications 899

Enterprise applications 899
Installing a new application 899
Preparing for application installation settings 904
Example: Installing an EAR file using the default bindings 908
Enterprise application collection 908
Name 908
Status 908
Enterprise application settings. 909
Starting and stopping applications 911
Exporting applications 912
Exporting DDL files 912
Updating applications 912
Hot deployment and dynamic reloading . . . 914
Uninstalling applications 922
Deploying and managing applications: Resources for learning 922

Notices 925

Trademarks and service marks. . . . 927

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Application Server Information Center
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Click on the **Feedback** link at the bottom of the article, and a separate window containing an e-mail form appears.
  3. Fill out the e-mail form as instructed, and click on **Submit feedback** .
- To send comments on PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com** or fax them to 919-254-0206.

  Be sure to include the document name and number, the WebSphere Application Server version you are using, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2003

Chapter 1. Welcome to Applications

The following items comprise the application programming model, including numerous services available to support deployed applications.

Web modules

Use Web components such as servlets and JavaServer Pages files to develop dynamic Web sites. Product extensions to the open source servlet and JSP APIs enhance standard features, and provide additional functionality.

Web modules consist of the following application components, each performing a different function:
- HTML and JSP pages provide the user interface and program logic
- Servlets coordinate work between other components of the application

HTTP sessions are a key area of product support for Web modules. By **managing HTTP sessions** for your Web applications, you can personalize a Web site for individual customers. A session is a series of requests to a servlet, originating from the same user at the same browser. Managing HTTP sessions allows servlets running in a Web container to keep track of individual users. For example, a servlet might use sessions to provide ″shopping carts″ to on-line shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she will purchase from the Web site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add choices of Shopper 1 to the cart of Shopper 2.

EJB modules

IBM WebSphere Application Server provides broad support for enterprise beans, including the Enterprise JavaBeans (EJB) 2.0 specification. The EJB 2.0 specification introduces a container-managed persistence (CMP) 2.0 component model, which provides a number of improvements to aid developer productivity and application performance. In addition, this product continues to fully support enterprise beans written to the CMP 1.1 programming model and deployed in previous versions of this product; applications can use CMP 1.1 beans, CMP 2.0 beans, or a mixture of both. CMP 1.1 beans can be directly carried forward in an EJB 1.1 ejb-jar module or may be repackaged and combined with CMP 2.0 beans in an EJB 2.0 module.

For EJB 2.0 modules, a feature introduced in Version 5 of this product, called **access intent** policies, eases the management of interactions between CMP beans and their underlying data stores. Each policy sets such data access characteristics such as access type (read or update) and transaction isolation that affect the locking of resources, letting you choose the level of data integrity and performance for your application.

Several excellent trade books that cover EJB 2.0 and the CMP 2.0 persistence model are already available. A good way to locate some of these is to visit your favorite online bookstore and search on the term *Enterprise JavaBeans*. For a more basic orientation, see “Enterprise beans: Resources for learning” on page 160.

© Copyright IBM Corp. 2003

Your application development might include **asynchronous messaging**, which the product supports as a method of communication based on the Java Message Service (JMS) programming interface.

The base JMS support enables IBM WebSphere Application Server applications to exchange messages asynchronously with other JMS clients by using JMS destinations (queues or topics). An application can explicitly poll for messages on a destination.

The product also provides a message listener service that applications can use to automatically retrieve messages from JMS destinations for processing by message-driven beans, without the application having to explicitly poll JMS destinations.

Client modules

For an overview, refer to Welcome to Client modules.

Web services

The Web services development and implementation components included in this product version are based on Apache SOAP 2.3. This information is deprecated in newer product versions.

5.0.2 + The Web services development and implementation components included with this product version are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE), Java for XML-based remote procedure call (JAX-RPC) and WS-I Basic Profile 1.0 specifications.

An open source implementation for a Web Services Invocation Framework (WSIF) is also supported.

Additional features, such as UDDI Registry are described in Welcome to Servers.

5.0.2 + WebSphere Application Server supports Web services security functionality that is based on standards included in the Web services security (WS-Security) specification.

Application services

IBM WebSphere Application Server provides essential services to ease the building of dynamic and flexible e-business applications. These services support and extend the open standards of J2EE and Web services, with a focus on application reuse and integration.

- **Class loading**

  The WebSphere Application Server product provides several class-loading modes, policies, and features to enable you to deploy and run your applications successfully. An application server provides an Application class-loader policy that enables you to control the isolation of applications in a server. If you want applications to share classes, choose the SINGLE policy; otherwise choose the MULTIPLE policy, which isolates the class loaders for each application.

  Similarly, at the application level, you can choose a WAR class-loader policy that configures the isolation of Web modules within an application. If you choose the policy APPLICATION, then each Web module in your application can see the

classes of other Web modules. A policy of MODULE creates a separate class loader for each Web module, resulting in isolation for each of the classes of each Web module.

The class-loader mode setting, which you can configure at the server, application, or Web module level depending on your class-loader policy, enables you to control whether application class loaders override classes contained in base run-time class loaders. By default, the WebSphere Application Server class loaders have a class-loader mode of PARENT_FIRST, which is the standard JDK mode and does not allow the application class loader to override classes. You must take care when using the PARENT_LAST class-loader mode to make all dependent classes available within the application or you might get LinkageErrors or other class-loader exceptions. For example, if you provide a newer version of the `Xerces.jar` file and your application is using XSLT, you must also provide a `xalan.jar` file within your application.

- **Shared library**

  Version 5.0 of WebSphere Application Server introduces the concept of a shared library. A shared library is a CLASSPATH and a symbolic name for that class path. You define shared libraries at the cell, node, or server level and then associate the shared libraries either with an application server (making the classes available to all applications in the server) or with individual applications (making the classes available only to the referencing application). This mechanism provides a convenient way to make libraries of classes available to your applications outside of a standard J2EE enterprise application (EAR) file for easier version management and space efficiency.
- **Internationalization support**

  If your application component must support multiple locales, the **localizable-text** API can help both developers and administrators through central management of displayed strings. The developer separates strings into a message catalog, which is then translated into the other languages required. All message catalogs are then deployed with the application component. From then on, the administrator can add or update message catalogs centrally as required. See Chapter 12, “Internationalizing applications,” on page 749.
- **Transactions**

  IBM WebSphere Application Server applications can use transactions to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent. The way that applications use transactions depends on the type of application component, as follows:
  - A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (where the bean manages transactions itself)
  - Entity beans use container-managed transactions
  - Web components (servlets) use bean-managed transactions

  The product is a transaction manager that supports the coordination of resource managers through their XAResource interface and participates in distributed global transactions with other OTS 1.2 compliant transaction managers (for example, J2EE 1.3 application servers). Applications can also be configured to interact with databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction coordination is not required.

  Resource managers that offer transaction support can be categorized into those that support two-phase coordination (by offering an XAResource interface) and those that support only one-phase coordination (for example through a LocalTransaction interface). The IBM WebSphere Application Server transaction

support provides coordination, within a transaction, for any number of two-phase capable resource managers. It also enables a single one-phase capable resource manager to be used within a transaction in the absence of any other resource managers, although a WebSphere transaction is not necessary in this case. With the Last Participant Support of Enterprise Extensions, you can coordinate the use of a single one-phase commit (1PC) capable resource with any number of two-phase commit (2PC) capable resources in the same global transaction. At transaction commit, the two-phase commit resources are prepared first using the two-phase commit protocol, and if this is successful the one-phase commit-resource is then called to `commit(one_phase)`. The two-phase commit resources are then committed or rolled back depending on the response of the one-phase commit resource.

- **Naming**

  Naming clients use **Naming Services** primarily to access objects, such as EJB homes, associated with applications installed on IBM WebSphere Application Server. Objects are made available to clients by being bound into a name space. A name space is under the control of a name server. In this product, there are potentially many name servers, and the name spaces controlled by the various name servers are federated together to form the view of a single name space. Each name server presents the same logical view of the federated name spaces.

  Name servers provided by this product are a CORBA CosNaming implementation. IBM WebSphere Application Server provides a CosNaming JNDI plug-in which enables clients to access the name servers through the JNDI interface. Clients to EJB applications typically use JNDI to perform Naming operations. Clients may access the name servers directly through the CORBA programming model. The CosNaming interface is part of the CORBA programming model. CORBA clients which need to access EJB homes or some other objects bound to the name space would typically use the CORBA CosNaming interface to perform Naming operations.

- **Dynamic cache**

  Dynamic cache improves application performance by caching outputs and contents of outputs of servlets, JavaServer Pages (JSP) files, Web services, and commands. On subsequent client requests to the same applications, dynamic cache intercepts these calls and responds by serving the output or the contents of output from the cache.

  Dynamic cache in this product version includes:

  Servlet/JSP files caching
  : This caches output of dynamic servlets and JSP files by working with Java virtual machine of the application server by intercepting calls to service methods and serving Web pages from the cache. This improves server response time, throughput and scalability.

  Command caching
  : Commands that are written to the Command Architecture encapsulate business logic tasks and provide a standard way to invoke the business logic request. Command objects need to implement CacheableCommand interface instead of TargetableCommand interface to cache. Like in servlets and JSP caching, requests to execute business logic in the command is intercepted by the cache. If a command with the same request attributes are available in cache, output properties are copied from the cached instance to the requested instance and returned without executing the business logic again.

  Web Services caching
  : Web service responses can be cached just like servlet and JSP results. These requests are intercepted and cache ID computed based on how the cache ID rules are specified in the cache policy. Hash of the whole

SOAPEnvelope can be used as a cache ID or it can be parsed and service name, operation name and parameter names to these operations used as cache ID. If a cache entry is not found for the computed cache ID, the request is forwarded to the SOAP engine and the result is cached.

Edge Side Include caching
: This provides the ability to cache, assemble and deliver dynamic web pages at the edge of the enterprise network. Edge Side Includes (ESI) is a simple markup language which enables dynamic web pages (which by themselves are not so cache efficient) to be broken down into cacheable fragments. These fragments are then cached on the edge of the network and assembled into a single page upon user requests.

Distributed caching
: Cache contents can be shared and replicated among servers by dynamic caching using an underlying JMS based message broker system, DRS (Data Replication Service). Sharing characteristics of individual cache entry is configured using the cache policy specification.

Assembly tools

Assembling is an activity in which you package code components into ″modules″ that comply with the J2EE specification. You define configurations for the modules, in the form of XML documents known as deployment descriptors.

5.0.2 See ″Chapter 17, “Assembling applications with the Assembly Toolkit,” on page 879″ or ″Chapter 16, “Assembling applications with the AAT,” on page 863.″

5.0.1 See ″Chapter 16, “Assembling applications with the AAT,” on page 863.″

Enterprise archive (EAR) files are comprised of the following archives:
- Enterprise bean (JAR) files (known as “EJB modules” on page 123)
- Web archive (WAR) files (known as “Web modules” on page 43)
- Application client (JAR) files (known as “Application clients” on page 189)
- Resource adapter archive (RAR) files (known as resource adapter modules)
- Additional JAR files containing dependent classes or other components required by the application

The standard file extension of an enterprise application file is .ear.

5.0.2 + For a discussion of archives and Web components supported by the Assembly Toolkit, see ″“Archive support in Version 5.0” on page 882.″

5.0.1 5.0.2 For a discussion of archives and Web components supported by the Application Assembly Tool in Version 5, see ″“Archive support in Version 5.0” on page 865.″

Deployment

Deployment involves placing applications onto application servers and running the applications. The main tasks include:
1. Installing application files onto an application server.
2. Configuring the application for the particular operational environment.
3. Starting the newly deployed application.

Information on these tasks is available from "Chapter 18, “Deploying and managing applications,” on page 899." The information describes how to deploy applications using the WebSphere Application Server administrative console. You can also deploy applications using the wsadmin tool, which provides deployment capabilities identical to those available using the administrative console.

Packaging and class loading

You can package your business logic as a Java 2 Platform, Enterprise Edition (J2EE) application enterprise archive (EAR) file or as an enterprise bean (EJB) or Web module for deployment to WebSphere Application Server. You must also consider the class loading relationships among modules.

Uninstalling and redeploying applications

At some point, you will need to uninstall your deployed applications. Or you might need to update your applications and deploy them again. You might be able to use hot deployment and dynamic reloading, where you do not need to restart the application server (or the application in some cases) after deploying an updated application.

Chapter 2. Using Web applications

A developer creates the files comprising a Web application, and then assembles the Web application components into a Web module. Next, the deployer (typically the developer in a unit-testing environment or the administrator in a production environment) installs the Web application on the server.

1. **(Optional)** Migrate existing Web applications to run in the new version of WebSphere.
2. Design the Web application and develop its code artifacts: Servlets, JavaServer Pages (JSP) files, and static files, as for example, images and Hyper Text Markup Language (HTML) files. See the "Resources for learning" article for links to design documentation.
3. **(Optional)** Implement JavaScript within JSP tags using the Bean Scripting Framework (BSF).
4. Develop the Web application, using WebSphere Application Server extensions to enhance its functionality.
5. Assemble the Web application into a Web module using theAssembly ToolkitorApplication Assembly Tool (AAT). Web module assembly properties might include the ability to:
   - Configure servlet page lists.
   - Configure servlet filters.
   - Serve servlets by class name.
   - Enable file serving.
6. Deploy the Web module or application module that contains the Web application.

   Following deployment, you might find it handy to use the tool that enables batch compiling of the JSP files for quicker initial response times.
7. **(Optional)** Troubleshoot your Web application.
8. **(Optional)** Modify the default Web container configuration in the application server in which you deployed the Web module or application module containing the Web application.
9. **(Optional)** Manage the deployed Web application..

Web applications

A Web application is comprised of one or more related servlets, JavaServer Pages technology (JSP files), and Hyper Text Markup Language (HTML) files that you can manage as a unit.

The files in a Web application are related in that they work together to perform a business logic function.

For example, one of the WebSphere Application Server samples is a *Simple Greeting* Web application. This application, comprised of a servlet and Web pages, greets new users when the application is accessed.

The Web application is a concept supported by the Java Servlet Specification. Web applications are typically packaged as `.war` files.

© Copyright IBM Corp. 2003

web.xml file

The web.xml file provides configuration and deployment information for the Web components that comprise a Web application. Examples of Web components are servlet parameters, servlet and JavaServer Pages (JSP) definitions, and Uniform Resource Locators (URL) mappings.

The servlet 2.3 specification dictates the format of the web.xml file, which makes this file portable among Java Two Enterprise Edition (J2EE) compliant products.

Location

The web.xml file must reside in the WEB-INF directory under the context of the hierarchy of directories that exist for a Web application. For example, if the application is client.war, then the web.xml file is placed in the *install_root*/*client war*/WEB-INF directory.

Usage notes

- Is this file read-only?

  No
- Is this file updated by a product component?

  This file is updated by the Assembly Toolkitor Application Assembly Tool (AAT).
- If so, what triggers its update?

  The Assembly Toolkitor AAT updates the web.xml file when you assemble Web components into a Web module, or when you modify the properties of the Web components or the Web module.
- How and when are the contents of this file used?

  WebSphere Application Server functions use information in this file during the configuration and deployment phases of Web application development.

Sample file entry

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.
        //DTD Web Application 2.3//EN"
               "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

   <web-app id="WebApp_1">
      <display-name>Persistence Manager Web Client</display-name>
      <description>Peristence Manager Web Client</description>
      <servlet id="Servlet_1">
        <servlet-name>CustomerLocalServlet</servlet-name>
         <description>Local Customer Servlet</description>
         <servlet-class>CustomerLocalServlet</servlet-class>
     </servlet>
      <servlet id="Servlet_2">
         <servlet-name>CustomerServlet</servlet-name>
        <description>Remote Customer Servlet</description>
         <servlet-class>CustomerServlet</servlet-class>
     </servlet>
     <servlet id="Servlet_3">
         <servlet-name>CreditCardServlet</servlet-name>
         <description>Credit Card Servlet - PM Verification</description>
         <servlet-class>CreditCardServlet</servlet-class>
      </servlet>
       <servlet-mapping id="ServletMapping_1">
        <servlet-name>CustomerLocalServlet</servlet-name>
        <url-pattern>/CustomerLocal</url-pattern>
      </servlet-mapping>
```

```
    <servlet-mapping id="ServletMapping_2">
      <servlet-name>CustomerServlet</servlet-name>
       <url-pattern>/Customer</url-pattern>
    </servlet-mapping>
    <servlet-mapping id="ServletMapping_3">
      <servlet-name>CreditCardServlet</servlet-name>
      <url-pattern>/CreditCard</url-pattern>
   </servlet-mapping>
   <welcome-file-list id="WelcomeFileList_1">
       <welcome-file>index.html</welcome-file>
   </welcome-file-list>
   <security-role id="SecurityRole_1">
       <description>Everyone role</description>
       <role-name>Everyone Role</role-name>
   </security-role>
    <security-role id="SecurityRole_2">
       <description>AllAuthenticated role</description>
       <role-name>All Role</role-name>
    </security-role>
    <security-role id="SecurityRole_3">
       <description>Deny all access role</description>
       <role-name>DenyAllRole</role-name>
    </security-role>
</web-app>
```

Migrating Web application components

Supported open specification levels in WebSphere Application Server Version 5 are documented in article, Migrating.

Migration of Web applications deployed in WebSphere Application Server Version 4.0.1 is not necessary; version 2.2 of the servlet specification and version 1.1 of the JavaServerPages (JSP) specification are still supported. However, where there are behavioral differences between the Java Two Enterprise Edition (J2EE) 1.2 and J2EE 1.3 specifications, bear in mind that J2EE 1.3 specifications are implemented in WebSphere Application Server Version 5 and will override any J2EE 1.2 behaviors.

Servlet migration might be a concern if your application:
- implements a WebSphere internal servlet to bypass a WebSphere Application Server Version 4.0.1 single application path restriction.
- extends a PageListServlet that relies on configuration information in the servlet configuration XML file.
- uses a servlet to generate Hyper Text Markup Language (HTML) output.
- calls the `response.sendRedirect()` method for a servlet using the `encodeRedirectURL` function or executing within a non-context root.

JSP migration might be a concern if your application references JSP page implementation classes in unnamed packages, or if you install WebSphere Application Server Version 4.0.1 EAR files (deployed in Version 4.0.1 with the JSP `Precompile` option), in Version 5.

Follow these steps if migration issues apply to your Web application:
1. Use WebSphere Application Server Version 5 package names for any WebSphere Application Server Version 4.0.1 internal servlets, which are implemented in your application.

   In WebSphere Application Server Version 4.0.1, Web modules with a context root setting of / are not supported. Accessing Web modules with this root context results in `HTTP 404 - File not Found` errrors.

To bypass the errors, and to enable the serving of static files from the root context, WebSphere Application Server Version 4.0.1 users are advised to add the servlet class, `com.ibm.servlet.engine.webapp.SimpleFileServlet`, to their Web module.

The Version 4.0.1 single path limitation does not exist in Version 5. However, users who choose to use the `com.ibm.servlet.engine.webapp.SimpleFileServlet` in Version 5 must do one of the following:

- Rename `com.ibm.servlet.engine.webapp.SimpleFileServlet` to `com.ibm.ws.webcontainer.servlet.SimpleFileServlet`.
- Opena Web deployment descriptor editor in the Assembly Toolkit and select **File serving enabled** on the **Extensions** tab. Or, open the EAR file in the Application Assembly Tool (AAT) and enable the `SimpleFileServlet` static file setting.

The following list identifies the other internal servlets affected by the Version 5 package name change:

- DefaultErrorReporter
- AutoInvoker

Use the Version 5 package name, `com.ibm.ws.webcontainer.servlet.`<*servlet class name*> for these servlets.

2. Migrate servlets that extend PageListServlet and rely on configuration information in the associated XML servlet configuration file. In Version 4.0.1, the XML servlet configuration file provides configuration data for page lists and augments servlet configuration information. This file is named as either *servlet_class_name*`.servlet` or *servlet_name*`.servlet`, and is stored in the same directory as the servlet class file.

   The XML servlet configuration file is not supported in WebSphere Application Server Version 5. The direct use of the servlet has been deprecated. The PageList servlet function is still available but is configured as part of the servlet extension configuration in the WAR file.

3. Set a content type if your servlet generates Hyper Text Markup Language (HTML) output.

   The default behavior of the Web container changed in WebSphere Application Server Version 5. If the servlet developer does not specify a content type in the servlet then the container is forbidden to set one automatically. Without an explicit content type setting, the content type is set to null. The Netscape browser displays HTML source as plain text with a null content type setting.

   To resolve this problem, do one of the following:

   - Explicitly set a content type in your servlet.
   - Opena Web deployment descriptor editor in the Assembly Toolkit and select **Automatic Response Encoding enabled** on the **Extensions** tab. Or, open the WAR file in the Application Assembly Tool (AAT) and enable the `autoResponseEncoding` static file setting.

4. Set the Java environment variable, `com.ibm.websphere.sendredirect.compatibility`, to **true** if you want your URLs interpreted relative to the application root.

   The default value of the Java environment variable `com.ibm.websphere.sendredirect.compatibility` changed in WebSphere Application Server Version 5. In Version 4, the default setting of this variable is true. In Version 5, the setting is false.

   When this variable is set to false, if a URL has a leading slash, the URL is interpreted relative to the Web module/application root. However, if the URL

does not have a leading slash, it is interpreted relative to the Web container root (also known as the Web server document root). Therefore, if an application has a WAR file that has a context root of myPledge_app and a servlet that has a servlet mapping of /Intranet/, a JSP file in the WAR file cannot access the servlet when its encodeRedirectURL is set to /Intranet/myPledge. The JSP file can access the servlet if the encodeRedirectURL is set to myPledge_app/Intranet/myPlege, or if the com.ibm.websphere.sendredirect.compatibility variable is set to true.

See the Setting the sendredirect variable article for more information.

5. Migrate WebSphere Version 4.0.1 enterprise applications to Version 5.

   Note: The WebSphere Application Server Version 4.0.1 JSP page implementation class files are not compatible with the WebSphere Application Server Version 5 JSP container.

   You must do one of the following:
   - Select the Pre-compile JSP option in the administrative console Install New Application window.

     See article Installing a new application for more information.
   - Specify the -preCompileJSPs option when using the Wsadmin tool.

6. Import your classes if your application uses unnamed packages.

   Section 8.2 of the JSP 1.2 specification states:

```
The JSP container creates a JSP page implementation class for each JSP page.
The name of the JSP page implementation class is implementation dependent.
The JSP page implementation object belongs to an implementation-dependent
named package. The package used may vary between one JSP and another, so
minimal assumptions should be made. The unnamed package should not be used
without an explicit import of the class.
```

   For example, if myBeanClass is in the unnamed package, and you reference it in a jsp:useBean tag, then you must explicitly import myBeanClass with the page directive import attribute, as shown in the following example:

```
<%@page import="myBeanClass" %>
              . . .
<jsp:useBean id="myBean" class="myBeanClass" scope="session"/>
```

   In WebSphere Application Server Version 5, the JSP engine creates JSP page implementation classes in the org.apache.jsp package. If a class in the unnamed package is not explicitly imported, then the javac compiler assumes the class is in package org.apache.jsp, and the compilation fails.

   Note: Avoid using the unnamed package altogether because of a change made in JDK 1.4 that will affect the JSP 2.0 specification. WebSphere Application Server Version 5 ships with JDK 1.3.1, so this is not an issue with the Version 5 JSP engine, but it will become an issue in future releases.

   The *Incompatibilities* section of the version 1.4.Java 2 Platform, Standard Edition (J2SE) documentation states:

```
     The compiler now rejects import statements that import a type from the
unnamed namespace. Previous versions of the compiler would accept such
import declarations, even though they were arguably not allowed by the
language (because the type name appearing in the import clause is not in
scope). The specification is being clarified to state clearly that you
cannot have a simple name in an import statement, nor can you import from
the unnamed namespace.
     To summarize, the syntax:

        import SimpleName;
```

```
is no longer legal. Nor is the syntax

        import ClassInUnnamedNamespace.Nested;

which would import a nested class from the unnamed namespace.
    To fix such problems in your code, move all of the classes from the
unnamed namespace into a named namespace.
```

See "Resources for learning" for links to the J2SE, JSP, and Servlet specification documentation.

Default Application

The IBM WebSphere Application Server provides a default configuration that allows administrators to easily verify that the Application Server is running. When the product is installed, it includes an application server called *server1* and an enterprise application called *Default Application*.

Default Application contains a Web Module called *DefaultWebApplication* and an enterprise bean JAR file called *Increment*. The *Default Application* provides a number of servlets, described below. These servlets are available in the product.

For additional code examples, visit the Samples Gallery. Learn how to locate and install the Samples Gallery by viewing the Samples Gallery reference page.

The URL for accessing Samples is: `http://localhost:9080/WSamples/`

Snoop

Use the Snoop servlet to retrieve information about a servlet request. This servlet returns the following information:
- Servlet initialization parameters
- Servlet context initialization parameters
- URL invocation request parameters
- Perferred client locale
- Context path
- User principal
- Request headers and their values
- Request parameter names and their values
- HTTPS protocol information
- Servlet request attributes and their values
- HTTP session information
- Session attributes and their values

The Snoop servlet includes security configuration so that when WebSphere Security is enabled, clients must supply a user ID and password to execute the servlet.

The URL for the Snoop servlet is: `http://localhost:9080/snoop/`.

HelloHTML

Use the HelloHTML pervasive servlet to exercise the PageList support provided by the WebSphere Web container. This servlet extends the PageListServlet, which provides APIs that allow servlets to call other Web resources by name or, when using the *Client Type detection* support, by type.

You can invoke the Hello servlet from an HTML browser, speech client, or most Wireless Application Protocol (WAP) enabled browsers using the URL: `http://localhost:9080/HelloHTML.jsp`.

HitCount

Use the HitCount Demonstration application to demonstrate incrementing a counter using a variety of methods, including:
- A servlet instance variable
- An HTTP session
- An enterprise bean

You can instruct the servlet to execute any of these methods within a transaction that you can ommit or roll back. If the transaction is committed, the counter is incremented. If the transaction is rolled back, the counter is not incremented.

The enterprise bean method uses a Container- Managed Persistence enterprise bean that persists the counter value to a Cloudscape database. This enterprise bean is configured to use the Default Datasource, which is set to the DefaultDB database.

When using the enterprise bean method, you can instruct the servlet to look up the enterprise bean, either in the WebSphere global namespace, or in the namespace local to the application.

The URL for the HitCount application is: `http://localhost:9080/HitCount.jsp`.

Servlets

Servlets are Java programs that use the Java Servlet Application Programming Interface (API). You must package servlets in a Web ARchive (WAR) file or Web module for deployment to the application server. *Servlets* run on a Java-enabled Web server and extend the capabilities of a Web server, similar to the way applets run on a browser and extend the capabilities of a browser.

Servlets can support dynamic Web page content, provide database access, serve multiple clients at one time, and filter data.

For the purposes of IBM WebSphere Application Server, discussions of servlets focus on Hyper Text Transfer Protocol (HTTP) servlets, which serve Web-based clients.

Developing servlets with WebSphere Application Server extensions

Several WebSphere Application Server extensions are provided for enhancing your servlets. This task provides a summary of the extensions that you can utilize.

1. Review the supported specifications.

   Create Java components, referring to the Servlet specifications from Sun Microsystems.

   See Resources for learning for links to coding specifications and examples.

   The application server includes its own packages that extend and add to the Java Servlet Application Programming Interface (API). These extensions and additions make it easier to manage session states, create personalized Web

pages, generate better servlet error reports, and access databases. Locate the Javadoc for the application server APIs in the product `install_root\web\apidocs` directory.

All the public WebSphere Application Server APIs are located in the `com.ibm.websphere...` packages.

2. Use your favorite integrated development environment (IDE), or a text editor, to develop or migrate code artifacts that meet the specifications.
3. Test the code artifacts.

Assemble your code artifacts into a Web module using theAssembly ToolkitorApplication Assembly Tool (AAT) as a prerequisite to deploying the code to the application server.

Application lifecycle listeners and events

Application lifecycle listeners and events, now part of the Servlet API, enable you to notify interested listeners when servlet contexts and sessions change. For example, you can notify users when attributes change and if sessions or servlet contexts are created or destroyed.

The lifecycle listeners give the application developer greater control over interactions with ServletContext and HttpSession objects. Servlet context listeners manage resources at an application level. Session listeners manage resources associated with a series of requests from a single client. Listeners are available for lifecycle events and for attribute modification events. The listener developer creates a class that implements the javax listener interface, corresponding to the desired listener functionality.

At application startup time, the container uses introspection to create an instance of your listener class and registers it with the appropriate event generator.

When a servlet context is created, the `contextInitialized` method of your listener class is invoked, which creates the database connection for the servlets in your application to use, if this context is for your application.

When the servlet context is destroyed, your `contextDestroyed` method is invoked, which releases the database connection, if this context is for your application.

Listener classes for servlet context and session changes

The following methods are defined as part of the `javax.servlet.ServletContextListener` interface:

- `void contextInitialized(ServletContextEvent)` - Notification that the Web application is ready to process requests.

  Place code in this method to see if the created context is for your Web application and if it is, allocate a database connection and store the connection in the servlet context.
- `void contextDestroyed(ServletContextEvent)` -Notification that the servlet context is about to shut down.

  Place code in this method to see if the created context is for your Web application and if it is, close the database connection stored in the servlet context.

Two new listener interfaces are defined as part of the javax.servlet package:
- ServletContextListener
- ServletContextAttributeListener

One new filter interface is defined as part of the javax.servlet package:
- FilterChain interface - methods: doFilter()

Two new event classes are defined as part of the javax.servlet package:
- ServletContextEvent
- ServletContextAttributeEvent

Three new listener interfaces are defined as part of the javax.servlet.http package:
- HttpSessionListener
- HttpSessionAttributeListener
- HttpSessionActivationListener

One new event class is defined as part of the javax.servlet.http package:
- HttpSessionEvent

Example: com.ibm.websphere.DBConnectionListener.java

The following example shows how to create a servlet context listener:

```
package com.ibm.websphere;

import java.io.*;
import javax.servlet.*;

public class DBConnectionListener implements ServletContextListener
{
     // implement the required context init method
    void contextInitialized(ServletContextEvent sce)
    {
    }

     // implement the required context destroy method
    void contextDestroyed(ServletContextEvent sce)
    {
    }
}
```

Servlet filtering

Servlet filtering is an integral part of the Servlet 2.3 API. Servlet filtering provides a new type of object called a *filter* that can transform a request or modify a response.

You can chain filters together so that a group of filters can act on the input and output of a specified resource or group of resources.

Filters typically include logging filters, image conversion filters, encryption filters, and Multipurpose Internet Mail Extensions (MIME) type filters (functionally equivalent to the servlet chaining). Although filters are not servlets, their lifecycle is very similar.

Filters are handled in the following manner:
- The Web container determines whether it needs to construct a `FilterChain` containing the `LoggingFilter` for the requested resource.

  The `FilterChain` begins with the invocation of the `LoggingFilter` and ends with the invocation of the requested resource.
- If other filters need to go in the chain, the Web container places them after the `LoggingFilter` and before the requested resource.
- The Web container then instantiates and initializes the `LoggingFilter` (if it was not done previously) and invokes its `doFilter(FilterConfig)` method to start the chain.

- The `LoggingFilter` preprocesses the request and response objects and then invokes the filter chain `doFilter(ServletRequest, ServletResponse)` method.

  This method passes the processing to the next resource in the chain (in this case, the requested resource).
- Upon return from the filter chain `doFilter(ServletRequest, ServletResponse)` method, the `LoggingFilter` performs post-processing on the request and response object before sending the response back to the client.

Filter, FilterChain, FilterConfig classes for servlet filtering

The following interfaces are defined as part of the javax.servlet package:

- Filter interface - methods: doFilter(), getFilterConfig(), setFilterConfig()
- FilterChain interface - methods: doFilter()
- FilterConfig interface - methods: getFilterName(), getInitParameter(), getInitParameterNames(), getServletContext()

The following classes are defined as part of the javax.servlet.http package:

- HttpServletRequestWrapper - methods: See the Servlet 2.3 Specification
- HttpServletResponseWrapper - methods: See the Servlet 2.3 Specification

Example: com.ibm.websphere.LoggingFilter.java

The following example shows how to implement a filter:

```
package com.ibm.websphere;

import java.io.*;
import javax.servlet.*;

public class LoggingFilter implements Filter
{
    File _loggingFile = null;

    // implement the required init method
    public void init(FilterConfig fc)
    {
        // create the logging file
        xxx;
    }

    // implement the required doFilter method...this is where most of
   the work is done
    public void doFilter(ServletRequest request,
     ServletResponse response, FilterChain chain)
    {
        try
        {
            // add request info to the log file
            synchronized(_loggingFile)
            {
                xxx;
            }

            // pass the request on to the next resource in the chain
            chain.doFilter(request, response);
        }
        catch (Throwable t)
        {
            // handle problem...
        }
    }

    // implement the required destroy method
    public void destroy()
```

```
    {
        // make sure logging file is closed
        _loggingFile.close();
    }
}
```

Configuring page list servlet client configurations

You can define PageListServlet configuration information in the IBM Web Extensions file. The IBM Web Extensions file is created and stored in the Web Applications archive (WAR) file by theAssembly Toolkitor Application Assembly Tool (AAT).

To configure and implement page lists:

1. To configure page list information, use the Add Markup Language entry dialog of the Assembly Toolkit. On the **Servlets** tab of a Web deployment descriptor editor, select a servlet and click **Add** under **WebSphere Extensions**.Or use the **PageList Extensions** tab in the AAT.
2. Add the `callPage()` method to your servlet to invoke a JavaServer Page (JSP) file in response to a client request.

   The PageListServlet has a `callPage()` method that invokes a JSP file in response to the HTTP request for a page in a page list. The `callPage()` method can be invoked in one of the following ways:
   - `callPage(String pageName, HttpServletRequest request, HttpServletResponse response)`

     where the method arguments are:

     pageName
     A page name defined in the PageListServlet configuration

     request
     The HttpServletRequest object

     response
     The HttpServletResponse object
   - `callPage(String mlName, String pageName, HttpServletRequest request, HttpServletResponse response)`

     where the method arguments are:

     mlName A markup language type

     pageName
     A page name defined in the PageListServlet configuration

     request
     The HttpServletRequest object

     response
     The HttpServletResponse object
3. Use the PageList Servlet client type detection support to determine the markup language type a calling client requires for the response.

Page lists

Page lists allow you to avoid hard-coding URLs in servlets and JSP files. A page list specifies the location where a request is to be forwarded, but automatically customizes that location depending on the MIME type of the servlet. Use these properties to specify a markup language and an associated MIME type. For the given MIME type, you also specify a set of pages to invoke.

WebSphere Application Server supplies the PageListServlet servlet, which you can use to call a JavaServer Pages (JSP) file by name based on the configuration data in the `client_types.xml` file. This file maps a JSP file to a Uniform Resource Identifier

(URI). When the URI is invoked, it specifies another JSP file in a Web module. This support allows you to access multiple Uniform Resource Locators (URLs) without hard-coding them in your servlets.

You can also logically group page lists according to the markup language type, such as, Hypertext Markup Language (HTML) or Wireless Markup Language (WML). This allows applications that use servlets to extend the PageListServlet servlet, to call JSP files which return the proper markup-language type for the client request. For example, a request that originates from a PDA device requires WML data. The application server sends the request to a servlet that extends the PageListServlet servlet, and the servlet calls a JSP file that returns a WML response.

Client type detection support

In addition to providing the page list mapping capability, the PageListServlet also provides *Client Type Detection* support. A servlet determines the markup language type that a calling client needs in the response, using the configuration information in the `client_types.xml` file.
Client type detection support allows a servlet, extending the PageListServlet, to call an appropriate JavaServer Pages (JSP) file. The servlet invokes the `callPage()` method, which calls a JSP file based on the markup-language type of the request.

client_types.xml

The `client_types.xml` file provides client type detection support for servlets extending PageListServlet. Using the configuration data in the `client_types.xml` file, servlets can determine the language type that calling clients require for the response.

The client type detection support allows servlets to call appropriate JavaServer Pages (JSP) files with the `callPage()` method. Servlets select JSP files based on the markup-language type of the request.

Servlets must use the following version of the `callPage()` method to determine the markup language type required by the client:

```
callPage(String mlName, String pageName, HttpServletRequest request,
                HttpServletResponse response)
```

where the arguments are:
- mlName - a markup language type
- pageName - a page name defined in the PageListServlet configuration
- request - the HttpServletRequest object
- response - the HttpServletResponse object

Review the Extending PageListServlet code example to see how the `callPage()` method is invoked by a servlet.

In the example, the client type detection method, `getMLTypeFromRequest(HttpServletRequestrequest)`, provided by the PageListServlet, inspects the HttpServletRequest object request headers, and searches for a match in the `client_types.xml` file.

The client type detection method does the following:
- Uses the input HttpServletRequest and the `client_types.xml` file, to check for a matching HTTP request name and value.
- Returns the markup-language value configured for the `<client-type>` element, if a match is found.

If multiple matches are found, this method returns the markup-language for the first `<client-type>` element for which a match is found.
- If no match is found, returns the value of the markup-language for the default page defined in the PageListServlet configuration.

Location

The `client_types.xml` file is located in the *install_root*/properties directory.

Usage notes
- Is this file read-only?

  No
- Is this file updated by a product component?

  No
- If so, what triggers its update?

  This file is created and updated manually by users.
- How and when are the contents of this file used?

  Servlets, extending PageListServlet, use this file to determine the language type that calling clients require for the response.

Sample file entry

```
<?xml version="1.0" >
<!DOCTYPE clients [
<!ELEMENT client-type (description, markup-language,request-header+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT markup-language (#PCDATA)>
<!ELEMENT request-header (name, value)>
<!ELEMENT clients (client-type+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT value (#PCDATA)>]>
<clients>
      <client-type>
      <description>IBM Speech Client</description>
      <markup-language>VXML</markup-language>
      <request-header>
           <name>user-agent</name>
           <value>IBM VoiceXML pre-release version 000303</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vxml</value>
      </request-header>
      </client-type>
      <client-type>
           <description>WML Browser</description>
           <markup-language>WML</markup-language>
      <request-header>
           <name>accept</name>
           <value>text/x-wap.wml</value>
      </request-header>
      <request-header>
           <name>accept</name>
           <value>text/vnd.wap.xml</value>
      </request-header>
    </client-type>
</clients>
```

Example: Extending PageListServlet

The following example shows how a servlet extends the PageListServlet class and determines the markup-language type required by the client. The servlet then uses

the callPage() method to call an appropriate JavaServer Pages (JSP) file. In this example, the JSP file that provides the the correct markup-language for the response is *Hello.page*.

```
public class HelloPervasiveServlet extends PageListServlet implements Serializable
{
     /*
     * doGet -- Process incoming HTTP GET requests
     */
    public  void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
       // This is the name of the page to be called:
       String pageName = "Hello.page";

       // First check if the servlet was invoked with a queryString that contains
    // a markup-language value.
       // For example, if this is how the  servlet is invoked:
       //  http://localhost/servlets/HeloPervasive?mlname=VXML
       //  then use the following method:
       String mlname= getMLNameFromRequest(request);

        // If no markup language type is provided in the queryString,
        // then try to determine the client
        // Type from the request, and use the markup-language name configured in
        // the client_types.xml file.
        if (mlName == null)
        {
          mlName = getMLTypeFromRequest(request);
         }
         try
         {
           // Serve the request page.
           callPage(mlName, pageName, request, response);
          }
          catch (Exception e)
          {
            handleError(mlName, request, response, e);
           }
       }
}
```

autoRequestEncoding and autoResponseEncoding

Two new WebSphere Application Server extensions are available in Version 5, autoRequestEncoding and autoResponseEncoding.

In WebSphere Application Server Version 5, the Web container no longer automatically sets request and response encodings, and response content types. Programmers are expected to set these values using available methods in the Servlet 2.3 Specification. If programmers choose not to use the character encoding methods, they can specify the autoRequestEncoding and autoResponseEncoding extensions, which enable the application server to set the encoding values and content type.

The values of the autoRequestEncoding and autoResponseEncoding extensions are either true or false. The default value for both extensions is false. If the value is false for both autoRequestEncoding and autoResponseEncoding, then the request and response character encoding is set to the Servlet 2.3 Specification default, which is ISO-8859-1. Also, If the value is set to false for a response, the Web container cannot set a response content type.

5.0.2 + Use the Assembly Toolkit to change the default values for the autoRequestEncoding and autoResponseEncoding extensions.

5.0.1 Use the Application Assembly Tool (AAT) to change the default values for the autoRequestEncoding and autoResponseEncoding extensions.

Review the autoRequestEncoding and autoResponseEncoding encoding examples for a description of Web container behavior when these values are set to true.

Examples: autoRequestEncoding and autoResponseEncoding encoding examples

The default value of the autoRequestEncoding and autoResponseEncoding extensions is false, which means that both the request and response character encoding is set to the Servlet 2.3 Specification default of ISO-8859-1. Different character encodings are possible if the client defines character encoding in the request header, or if the code includes the setCharacterEncoding(String encoding) method. Also, If the value is set to false for a response, the Web container cannot set a response content type.

If the autoRequestEncoding value is set to true, and the client did not specify character encoding in the request header, and the code does not include the setCharacterEncoding(String encoding) method, the Web container tries to determine the correct character encoding for the request parameters and data.

The Web container performs each step in the following list until a match is found:
- Looks at the character set (charset) in the *Content-Type* header.
- Attempts to map the servers locale to a character set using defined properties.
- Attempts to use the DEFAULT_CLIENT_ENCODING system property, if one is set.
- Uses the ISO-8859-1 character encoding as the default.

If the autoResponsetEncoding value is set to true, and the client did not specify character encoding in the request header, and the code does not include the setCharacterEncoding(String encoding) method, the Web container does the following:
- Attempts to determine the response content type and character encoding from information in the request header.
- Uses the ISO-8859-1 character encoding as the default.

JavaServer Pages files

JavaServer Pages (JSP) files are application components coded to the Sun Microsystems JavaServer Pages (JSP) Specification. JSP files enable the separation of the Hypertext Markup Language (HTML) code from the business logic in Web pages so that HTML programmers and Java programmers can more easily collaborate in creating and maintaining pages.

The IBM extensions to the JSP Specification include JSP tags that resemble HTML tags. These JSP tags make it easy for HTML authors to add the power of Java technology to Web pages, without being experts in Java programming.

JSP files support a division of roles:

HTML authors
: Develop JSP files that access databases and reusable Java components, such as servlets and beans.

Java programmers
: Create the reusable Java components and provide the HTML authors with the component names and attributes.

Database administrators
Provide the HTML authors with the name of the database access and table information.

Developing JavaServer Pages files with WebSphere extensions

Several IBM WebSphere extensions are provided for enhancing your JavaServer Pages (JSP) files. This task provides a summary of the extensions that you can utilize.

1. Review the supported specifications.

   Create Java components, referring to the JSP specifications from Sun Microsystems.

   See Resources for learning for links to coding specifications and examples.

   WebSphere Application Server Version 3.5 added IBM extensions to the base Application Programming Interfaces (APIs). Since the JavaServer Pages (JSP) 1.1 and JSP 1.2 Specifications are backward compatible to the JSP 1.0 Specifications, you can invoke the APIs with the IBM extensions without modification.

   The extensions belong to these categories:

   Syntax for variable data
   Put variable fields in JSP files and have servlets and beans dynamically replace the variables with values from a database when the JSP output is returned to the browser.

   Syntax for database access
   Add a database connection to a Web page and then use that connection to query or update the database. You can provide the user ID and password for the database connection at request time, or you can hard code the user ID and password within the JSP file.
2. Use your favorite integrated development environment (IDE), or a text editor, to develop or migrate code artifacts that meet the specifications.
3. Test the code artifacts.
4. **(Optional)** Batch compile your JSP files if necessary.

Tag libraries

Java ServerPages (JSP) tag libraries contain classes for common tasks such as processing forms and accessing databases from JSP files.

Tag libraries encapsulate, as simple tags, core functionality common to many Web applications. The Java Standard Tag Library (JSTL) supports common programming tasks such as iteration and conditional processing, and provides tags for:

- manipulating XML documents
- supporting internationalization
- using Structured Query Language (SQL)

Tag libraries also introduce the concept of an expression language to simplify page development, and include a version of the JSP expression language.

A tag library has two parts - a Tag Library Descriptor (TLD) file and a JAR file.

tsx:dbconnect tag JavaServer Pages syntax

Use the <tsx:dbconnect> tag to specify information needed to make a connection to a Java Database Connectivity (JDBC) or an Open Database Connectivity (ODBC) database.

The <tsx:dbconnect> syntax does not establish the connection. Use the <tsx:dbquery> and <tsx:dbmodify> syntax instead to reference a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file to establish the connection.

When the JSP file compiles into a servlet, the Java processor adds the Java coding for the <tsx:dbconnect> syntax to the servlet service() method, which means a new database connection is created for each request for the JSP file.

This section describes the syntax of the <tsx:dbconnect> tag.

```
<tsx:dbconnect id="connection_id"
    userid="db_user" passwd="user_password"
    url="jdbc:subprotocol:database"
    driver="database_driver_name"
    jndiname="JNDI_context/logical_name">
</tsx:dbconnect>
```

where:

- **id**

  Represents a required identifier. The scope is the JSP file. This identifier is referenced by the connection attribute of a <tsx:dbquery> tag.
- **userid**

  Represents an optional attribute that specifies a valid user ID for the database that you want to access. Specify this attribute to add the attribute and its value to the request object.

  Although the userid attribute is optional, you must provide the user ID. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this information in the JSP file.
- **passwd**

  Represents an optional attribute that specifies the user password for the userid attribute. (This attribute is not optional if the userid attribute is specified.) If you specify this attribute, the attribute and its value are added to the request object.

  Although the passwd attribute is optional, you must provide the password. See <tsx:userid> and <tsx:passwd> for an alternative to hard coding this attribute in the JSP file.
- **url** and **driver**

  Respresents a required attribute if you want to establish a database connection. You must provide the URL and driver.

  The application server supports connection to JDBC databases and ODBC databases.

  – For a JDBC database, the URL consists of the following colon-separated elements: jdbc, the subprotocol name, and the name of the database to access. An example for a connection to the Sample database included with IBM DB2 is:

    ```
    url="jdbc:db2:sample"
    driver="COM.ibm.db2.jdbc.app.DB2Driver"
    ```

  – For an ODBC database, use the Sun JDBC-to-ODBC bridge driver included in their Java2 Software Developers Kit (SDK) or another vendor's ODBC driver.

    The url attribute specifies the location of the database. The driver attribute specifies the name of the driver to use in establishing the database connection.

    If the database is an ODBC database, you can use an ODBC driver or the Sun JDBC-to-ODBC bridge. If you want to use an ODBC driver, refer to the driver documentation for instructions on specifying the database location with the url attribute and the driver name.

If you use the bridge, the url syntax is `jdbc:odbc:database`. An example follows:

```
url="jdbc:odbc:autos"
driver="sun.jdbc.odbc.JdbcOdbcDriver"
```

Note: To enable the application server to access the ODBC database, use the ODBC Data Source Administrator to add the ODBC data source to the System DSN configuration. To access the ODBC Administrator, click the ODBC icon on the Windows NT Control Panel.

- **jndiname**

  Represents an optional attribute that identifies a valid context in the application server Java Naming and Directory Interface (JNDI) naming context and the logical name of the data source in that context. The Web administrator configures the context using an administrative client such as the WebSphere Administrative Console.

  If you specify the jndiname attribute, the JSP processor ignores the driver and url attributes on the <tsx:dbconnect> tag.

An empty element (such as <url></url>) is valid.

dbquery tag JavaServer Pages syntax

Use the <tsx:dbquery> tag to establish a connection to a database, submit database queries, and return the results set.

The <tsx:dbquery> tag does the following:

1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information the tag provides to determine the database URL and driver. You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are provided in the <tsx:dbconnect> tag.
2. Establishes a new connection
3. Retrieves and caches data in the results object.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbquery> tag.

```
<%-- SELECT commands and (optional) JSP syntax can be placed within the tsx:dbquery. --%>
<%-- Any other syntax, including HTML comments, are not valid. --%>
<tsx:dbquery id="query_id" connection="connection_id" limit="value" >
</tsx:dbquery>
```

where:

- **id**

  Represents the identifier of this query. The scope is the JSP file. Use `id` to reference the query. For example, from the <tsx:getProperty> tag, use `id` to display the query results.

  The `id` is a tsx reference to the bean and can be used to retrieve the bean from the page contect. For example, if id is named mySingleDBBean, instead of using:

  – if (mySingleDBBean.getValue(″UISEAM″,0).startsWith(″N″))

  use:

  – com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults bean = (com.ibm.ws.webcontainer.jsp.tsx.db.QueryResults)pageContext. findAttribute(″mySingleDBBean″); if (bean.getValue(″UISEAM″,0).startsWith(″N″)). . .

The bean properties are dynamic and the property names are the names of the columns in the results set. If you want different column names, use the SQL keyword for specifying an alias on the SELECT command. In the following example, the database table contains columns named FNAME and LNAME, but the SELECT statement uses the AS keyword to map those column names to FirstName and LastName in the results set:

```
Select FNAME, LNAME AS FirstName, LastName from Employee where FNAME='Jim'
```

- **connection**

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the database URL, driver name, and optionally, the user ID and password for the connection.
- **limit**

  Represents an optional attribute that constrains the maximum number of records returned by a query. If this attribute is not specified, no limit is used. In such a case, the effective limit is determined by the number of records and the system caching capability.
- SELECT command and JSP syntax

  Represents the only valid SQL command, SELECT. The <tsx:dbquery> tag must return a results set. Refer to your database documentation for information about the SELECT command. See other articles in this section for a description of JSP syntax for variable data and inline Java code.

dbmodify tag JavaServer Pages syntax

The <tsx:dbmodify> tag establishes a connection to a database and then adds records to a database table.

The <tsx:dbmodify> tag does the following:

1. References a <tsx:dbconnect> tag in the same JavaServer Pages (JSP) file and uses the information provided by that tag to determine the database URL and driver.

   Note: You can also obtain the user ID and password from the <tsx:dbconnect> tag if those values are provided in the <tsx:dbconnect> tag.
2. Establishes a new connection.
3. Updates a table in the database.
4. Closes the connection and releases the connection resource.

This section describes the syntax of the <tsx:dbmodify> tag.

```
<%-- Any valid database update commands can be placed within the DBMODIFY tag. -->
<%-- Any other syntax, including HTML comments, are not valid. -->
<tsx:dbmodify connection="connection_id">
</tsx:dbmodify>
```

where:

- **connection**

  Represents the identifier of a <tsx:dbconnect> tag in this JSP file. The <tsx:dbconnect> tag provides the database URL, driver name, and (optionally) the user ID and password for the connection.
- Database commands

  Represents valid database commands. Refer to your database documentation for details

tsx:getProperty tag JavaServer Pages syntax and examples

The <tsx:getProperty> tag gets the value of a bean to display in a JavaServer Pages (JSP) file.

This IBM extension of the Sun JSP <jsp:getProperty> tag implements all of the <jsp:getProperty> function and adds the ability to introspect a database bean created using the IBM extension <tsx:dbquery> or <tsx:dbmodify>.

Note: You cannot assign the value from this tag to a variable. The value, generated as output from this tag, displays in the browser window.

This section describes the syntax of the <tsx:getProperty> tag:

```
<tsx:getProperty name="bean_name"
  property="property_name" />
```

where:

- **name**

  Represents the name of the bean declared by the id attribute of a <tsx:dbquery> syntax within the JSP file. See <tsx:dbquery> for an explanation. The value of this attribute is case-sensitive.
- **property**

  Represents the property of the bean to access for substitution. The value of the attribute is case-sensitive and is the locale-independent name of the property.

Tag example:

```
<tsx:getProperty name="userProfile" property="username" />
<tsx:getProperty name="request" property=request.getParameter("corporation") />
```

In most cases, the value of the property attribute is just the property name. However, to access the request bean or to access a property of a property (sub property), specify the full form of the property attribute. The full form also gives you the option to specify an index for indexed properties. You can specify the optional index as a constant (such as 2), or an index like the one described in the <tsx:repeat> tag. Some examples using the full form of the property attribute follow:

```
<tsx:getProperty name="staffQuery" property=address(currentAddressIndex) />
<tsx:getProperty name="shoppingCart" property=items(4).price />
<tsx:getProperty name="fooBean" property=foo(2).bat(3).boo.far />
```

tsx:userid and tsx:passwd tag JavaServer Pages syntax

With the <tsx:userid> and <tsx:passwd> tags, you do not have to hard code a user ID and password in the <tsx:dbconnect> tag.

Use the <tsx:userid> and <tsx:passwd> tags to accept user input for the values and then add that data to the request object. You can access the request object with a JavaServer Pages (JSP) file, such as the *JSPEmployee.jsp* example that requests the database connection.

You must use <tsx:userid> and <tsx:passwd> tags within a <tsx:dbconnect> tag.

This section describes the syntax of the <tsx:userid> and <tsx:passwd> tags.

```
<tsx:dbconnect id="connection_id"
    <font color="red"><userid></font>
    <tsx:getProperty name="request" property=request.getParameter("userid") />
    <font color="red"></userid></font>
    <font color="red"><passwd></font>
    <tsx:getProperty name="request" property=request.getParameter("passwd") />
```

```
    <font color="red"></passwd></font>
    url="protocol:database_name:database_table"
    driver="JDBC_driver_name">
</tsx:dbconnect>
```

where:

- **<tsx:getProperty>**

  Represents the syntax as a mechanism for embedding variable data.
- **userid**

  Represents a reference to the request parameter that contains the user ID. You must add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string to pass the user-specified request parameters.
- **passwd**

  Represents a reference to the request parameter that contains the password. Add the parameter to the request object that passes to this JSP file. You can set the attribute and its value in the request object, using an HTML form or a URL query string, to pass user-specified values.

tsx:repeat tag JavaServer Pages syntax

The <tsx:getProperty> tag repeats a block of HTML tagging.

Use the <tsx:repeat> syntax to iterate over a database query results set. The <tsx:repeat> syntax iterates from the start value to the end value until one of the following conditions is met:

- The end value is reached.
- An exception is thrown.

The output of a <tsx:repeat> block is buffered until the block completes. If an exception is thrown before a block completes, no output is written for that block.

This section describes the syntax of the `<tsx:repeat>` tag:

```
<tsx:repeat index=name start="starting_index" end="ending_index">
</tsx:repeat>
```

where:

- **index**

  Represents an optional name used to identify the index of this repeat block. The value is case-sensitive and its scope is the JSP file.
- **start**

  Represents an optional starting index value for this repeat block. The default is `0`.
- **end**

  Represents an optional ending index value for this repeat block. The maximum value is `2,147,483,647`.

  If the value of the end attribute is less than the value of the start attribute, the end attribute is ignored.

Example: Combining tsx:repeat and tsx:getProperty JavaServer Pages tags

The following code snippet shows you how to code these tags:

```
<tsx:repeat>
<tr>
    <td><tsx:getProperty name="empqs" property="EMPNO" />
```

```
    <tsx:getProperty name="empqs" property="FIRSTNME" />
    <tsx:getProperty name="empqs" property="WORKDEPT" />
    <tsx:getProperty name="empqs" property="EDLEVEL" />
    </td>
</tr>
</tsx:repeat>
```

Example: tsx:dbmodify tag syntax

In the following example, a new employee record is added to a database. The values of the fields are based on user input from this JavaServer Pages (JSP) file and referenced in the database commands using the <tsx:getProperty> tag.

```
<tsx:dbmodify connection="conn" >
insert into EMPLOYEE
    (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,EDLEVEL)
values
('<tsx:getProperty name="request" property=request.getParameter("EMPNO") />',
'<tsx:getProperty name="request" property=request.getParameter("FIRSTNME") />',
'<tsx:getProperty name="request" property=request.getParameter("MIDINIT") />',
'<tsx:getProperty name="request" property=request.getParameter("LASTNAME") />',
'<tsx:getProperty name="request" property=request.getParameter("WORKDEPT") />',
<tsx:getProperty name="request" property=request.getParameter("EDLEVEL") />)
</tsx:dbmodify>
```

Example: Using tsx:repeat JavaServer Pages tag to iterate over a results set

The <tsx:repeat> tag iterates over a results set. The results set is contained within a bean. The bean can be a static bean, for example, a bean created by using the IBM WebSphere Studio database wizard, or a dynamically generated bean, for example, a bean generated by the <tsx:dbquery> syntax. The following table is a graphic representation of the contents of a bean called, *myBean*:

	col1	col2	col3
row0	friends	Romans	countrymen
row1	bacon	lettuce	tomato
row2	May	June	July

Some observations about the bean:
- The column names in the database table become the property names of the bean. The <tsx:dbquery> section describes a technique for mapping the column names to different property names.
- The bean properties are indexed. For example, `myBean.get(Col1(row2))` returns `May`.
- The query results are in the rows. The <tsx:repeat> tag iterates over the rows, beginning at the start row.

The following table compares using the <tsx:repeat> tag to iterate over a static bean, versus a dynamically generated bean:

Static Bean Example	<tsx:repeat> Bean Example

myBean.class
// Code to get a connection
 	
// Code to get the data
Select * from myTable;
 	
// Code to close the connection
 	
JSP file
<tsx:repeat index=abc>
<tsx:getProperty name="myBean"
property="col1(abc)" />
</tsx:repeat>
 	
Notes:
• The bean (myBean.class) is a static bean.
• The method to access the bean properties is myBean.get(*property*(*index*)).
• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
• The <tsx:repeat> tag iterates over the bean properties row by row, beginning with the start row.
	JSP file
	<tsx:dbconnect id="conn"
	userid="alice"passwd="test"
	url="jdbc:db2:sample"
	driver="COM.ibm.db2.jdbc.app.DB2Driver">
	</tsx:dbconnect >
	
	<tsx:dbquery id="dynamic"
	connection="conn" >
	Select * from myTable;
	</tsx:dbquery>
	
	<tsx:repeat index=abc>
	<tsx:getProperty name="dynamic"
	property="col1(abc)" />
	</tsx:repeat>
	
	Notes:
	• The bean (dynamic) is generated by the <tsx:dbquery> tag and does not exist until the syntax executes.
	• The method to access the bean properties is dynamic.getValue("*property*", *index*).
	• You can omit the property index, in which case the index of the enclosing <tsx:repeat> tag is used. You can also omit the index on the <tsx:repeat> tag.
	• The <tsx:repeat> tag syntax iterates over the bean properties row by row, beginning with the start row.

Implicit and explicit indexing

Examples 1, 2, and 3 show how to use the <tsx:repeat> tag. The examples produce the same output if all indexed properties have 300 or fewer elements. If there are more than 300 elements, Examples 1 and 2 display all elements, while Example 3 shows only the first 300 elements.

Example 1 shows *implicit indexing* with the default start and default end index. The bean with the smallest number of indexed properties restricts the number of times the loop repeats.

```
<table>
<tsx:repeat>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone" />
  </tr></td>
</tsx:repeat>
</table>
```

Example 2 shows indexing, starting index, and ending index:

```
<table>
<tsx:repeat index=myIndex start=0 end=2147483647>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=city(myIndex) />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=address(myIndex) />
  </tr></td>
```

```
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property=telephone(myIndex) />
</tr></td>
</tsx:repeat>
</table>
```

Example 3 shows *explicit indexing* and ending index with implicit starting index. Although the index attribute is specified, you can still implicitly index the indexed property city because the `(myIndex)` tag is not required.

```
<table>
<tsx:repeat index=myIndex end=299>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="city" /t>
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="address(myIndex)" />
  </tr></td>
  <tr><td><tsx:getProperty name="serviceLocationsQuery" property="telephone(myIndex)" />
  </tr></td>
</tsx:repeat>
</table>
```

Nesting <tsx:repeat> blocks

You can nest <tsx:repeat> blocks. Each block is separately indexed. This capability is useful for interleaving properties on two beans, or properties that have subproperties. In the example, two <tsx:repeat> blocks are nested to display the list of songs on each compact disc in the user's shopping cart.

```
<tsx:repeat index=cdindex>
  <h1><tsx:getProperty name="shoppingCart" property=cds.title /></h1>
  <table>
  <tsx:repeat>
    <tr><td><tsx:getProperty name="shoppingCart" property=cds(cdindex).playlist />
    </td></tr>
  </tsx:repeat>
  </table>
 </tsx:repeat>
```

JspBatchCompiler tool

As an IBM enhancement to JavaServer Pages support, IBM WebSphere Application Server provides a batch JSP compiler. Use this function to batch compile your JSP files and thereby enable faster responses to the initial client requests for the JSP files on your production Web server.

Batch compiling makes the first request for a JSP file much faster because the JSP file is translated and compiled into a servlet. Batch compiling is also useful as a fast way to resynchronize all of the JSP files for an application.

To use the JSP batch compiler for JSP files, enter the following command on a single line at an operating system command prompt:

```
JspBatchCompiler -enterpriseapp.name <name>
                              [ -webmodule.name <name>]
                              [ -cell.name  <name>]
                              [ -node.name  <name>]
                              [ -server.name  <name>]
                              [ -filename <jsp name>]
                              [ -keepgenerated <true|false>]
                              [ -verbose <true|false>]
                              [ -deprecation <true|false>]
```

If the names specified for these arguments are comprised of two or more words separated by spaces, you must add quotation marks around the names.

where:

- **enterpriseapp.name**

  Represents the name of the enterprise application you want to compile.
- **webmodule.name**

  Represents the name of the specific Web module that you want to compile. If this argument is not set, all Web modules in the enterprise application are compiled.
- **cell.name**

  Represents the name of the cell in which the application is deployed. The default is `BaseApplicationServerCell`.
- **node.name**

  Represents the name of the node in which the application is deployed. The default is `DefaultNode`.
- **server.name**

  Represents the name of the server in which the application is deployed. The default is `server1`.
- **filename**

  Represents the name of a single JSP file that you want to compile. If this argument is not set, all files in the Web module are compiled. Alternatively, if *filename* is set to the name of a directory, only the JSP files in that directory are compiled.
- **keepgenerated**

  Represents the option to save or erase the generated files.

  If set to `yes`, WebSphere Application Server saves the generated `.java` files used for compilation on your server. By default, this argument is set to `no` and the `.java` files are erased after the class files have compiled.
- **verbose**

  Indicates the compiler should generate verbose output while compiling the generated sources.
- **deprecation**

  Indicates the compiler should generate deprecation warnings while compiling the generated sources.

Bean Scripting Framework

The Bean Scripting Framework (BSF) enables you to use scripting language functions in your Java server-side applications. This framework also extends scripting languages so that you can use existing Java classes and Java beans in the JavaScript language.

With BSF, you can write scripts that create, manipulate and access values from Java objects, or you can write Java programs that evaluate and access results from scripts.

WebSphere Application Server provides the Bean Scripting Framework, which consists of a BSF manager, a BSF engine, and a scripting engine.

BSF provides an access mechanism to Java objects for the scripting languages it supports, so that both the scripting language and theJava code can access code exclusive functions. The access mechanism is implemented through a registry of objects maintained by BSF.

BSF in WebSphere Application Server supports the Rhino ECMAScript.

The "Resources for Learning" article provides external BSF links that document future supported languages.

Developing Web applications

Design a Web application and the components that it needs.

For general Web application design information, see "Resources for learning."

There are two basic approaches to selecting tools for developing Web applications:
- You can use one of the available integrated development environments (IDEs). IDE tools automatically generate significant parts of the servlet and JavaServer Pages (JSP) code, and Hypertext Markup Language (HTML) files. They also contain integrated tools for packaging and testing the Web application components. The IBM WebSphere Application Developer product is the recommended IDE. For more information, see the documentation for that product.
- If you decide to develop Web components without an IDE, you need at least an ASCII text editor. You can also use tools available in the Java Software Development Kit (SDK) and in this product to assemble, test, and deploy the Web application components.

The following steps support the second approach, development without an IDE.
1. If necessary, migrate any pre-existing code to the required version of the servlet and JSP specification.
2. Write and compile the components of the Web application. To access classes that were extended, compile your code using the `-classpath` option on the `javac` compiler. This option allows you to reference the `j2ee.jar` file in the product `<install_root>\lib` directory.

   For example, to compile a servlet running on the Windows NT version of WebSphere Application Server, specify:

```
javac -classpath D:\Program Files\WebSphere\AppServer\lib\j2ee.jar MyServlet.java
```

   To compile that same servlet on the Windows NT version of WebSphere Network Deployment, specify:

```
javac -classpath D:\Program Files\WebSphere\DeploymentManager\lib\j2ee.jar MyServlet.java
```

3. 5.0.1 + **(Optional)** Disable JavaServer Pages (JSP) runtime compilation, if necessary.

Assemble the application components in one or more Web modules.

Disabling JavaServer Pages run-time compilation

By default, the JavaServer Pages (JSP) engine translates a requested JSP file, compiles the `.java` file, and loads the compiled servlet into the run-time environment. In previous releases of WebSphere Application Server, if a `.class` file did not exist, the JSP engine always translated and compiled the JSP file. You had to turn off the Web applications reload capability to prevent additional translations and recompiles of the file.

With Version 5.0.1 of WebSphere Application Server, you can now change the JSP engine default behavior by indicating a JSP file should never be translated or compiled at run time, even when a `.class` file does not exist.

If run-time compilation is disabled, you must precompile the JSP files, which provides the following advantages:
- Reduces compilation related disk operations.
- Minimizes disk storage requirements necessary for handling temporary `.java` and `.class` files generated during a run-time compilation.
- Forces you to verify that a JSP file compiled successfully before deploying and installing the application in WebSphere Application Server.

You can disable run-time JSP file compilation on a global or an individual Web application basis:
- To disable the translation and compilation of JSP files for all Web applications, set the Web container Custom property `disableJspRuntimeCompilation` to `true`.

  Set this property through the Web container `Custom properties` panel in the administrative console. To view this administrative console page, click:

  Servers > **Application Servers** > *server_name* > **Web Container** > **Custom Properties** > *property_name*

  Valid values for this setting are `true` or `false`. If this property is set to `true`, then translation and compilation of the JSP files is disabled at run time for all Web applications.
- To disable the translation and compilation of JSP files for a specific Web application, set the JSP engine initialization parameter `disableJspRuntimeCompilation` to `true`. This setting, if enabled, determines the run-time behavior of the JSP engine and overrides the Web container custom property setting.

  Set this parameter through the `JavaServer Pages attribute assembly settings` panel in the Application Assembly Tool (AAT)or . To view this page in the AAT, click:

  Web Modules > *component_instance* > **Assembly Property Extensions**

  Valid values for this setting are `true` or `false`. If this parameter is set to `true`, then, for that specific Web application, translation and compilation of the JSP files is disabled at run time, and the JSP engine only loads precompiled files.
- If neither the Web container custom property nor the JSP attribute assembly parameter is set, the first request for a JSP file results in the translation and compilation of the JSP file when the `.class` file does not exist. Subsequent requests for the file also result in compilations and translations, but only if the following conditions are met:
  - Compilations and translations are required.
  - Reloading is enabled for the Web module.
  - Reload interval is exceeded.

If you disable run-time compilation and a request arrives for a JSP file that does not have a matching `.class` file, the JSP engine returns HTTP error 501 (Not implemented) to the browser. If the JSP file does not exist, the JSP engine returns HTTP error 404 (File not found) to the browser. In both cases, an exception is written to the joblog (sysprint) file if ras_trace_outputLocation in `was.env` file is set to SYSPRINT or to CTRACE if ras_trace_outputLocation is set to BUFFER. If a JSP file has a matching `.class` file but that file is out of date, the JSP engine still loads the `.class` file into memory.

Perform the following steps to determine whether the `disableJspRuntimeCompilation` option is enabled in WebSphere Application Server:
1. Enable the Diagnostic Trace Service and set the trace specification to `com.ibm.ws.webcontainer.jsp.servlet.*=all=enabled`.

2. Request a JSP file.
3. Locate the string, `disableJspRuntimeCompilation:true`, in the joblog (sysprint) file if ras_trace_outputLocation in `was.env` file is set to SYSPRINT or to CTRACE if ras_trace_outputLocation is set to BUFFER.
4. Ensure the `jspUri:` entry matches the requested JSP file.

If both the `disableJspRuntimeCompilation:true` string and the matching `jspUri:` entry appear in the trace, the `disableJspRuntimeCompilation` setting is enabled for the Web application.

Example: Converting JavaScript source to the Bean Scripting Framework

JavaScript code is one of the most popular languages of Web developers. This language supports the following base objects, plus additional objects from the Document Object Model:
- array
- date
- math
- number
- string

Server-side JavaScript code supports the same base objects, and additional objects that support user access to databases, file systems and e-mail systems.

Like client-side JavaScript code, server-side JavaScript code is also platform, browser, and language independent.

You can convert server-side JavaScript applications to the Bean Scripting Framework. This article describes how to perform this conversion.

Server-side JavaScript source code

Suppose you have the following server-side JavaScript application:

```
<html>
<head>
<title>Hello World server-side JavaScript example</title>
</head>
<body>
<br><br>
</body>
</html>

<server>
function writePage()
  write("<center><font size='6'>Hello World</font></center>");
</server>
```

Converting server-side JavaScript source code to the Bean Scripting Framework (BSF)

Make the following changes to the JavaScript source code to enable BSF:

```
<%@  page language="javascript" %>
<html>
<head>
<title>Hello World server-side BSF/JavaScript example</title>
</head>
<body>
```

```
<br><br>
</body>
</html>

<%
  out.println("<center><font size='6'>Hello World</font></center>");
%>
```

Review the other BSF reference articles for deployment information and additional programming examples.

Scenario: Creating a Bean Scripting Framework application

Scenario description

Programming skills in JavaScript code are more prevalent than programming skills using JavaServer Pages (JSP) tags. Using the Bean Scripting Framework, JavaScript programmers can gradually introduce JSP tags in their JavaScript applications without completely rewriting the source code. The BSF method not only reduces the potential of programming errors, but also provides a painless way to learn a new technology.

The following scenario illustrates how to implement a BSF application using JavaScript within JSP tags.

Developing the BSF application

At ABC elementary school, John Doe teaches third grade mathematics. He wants to help his students memorize their multiplication tables, and thinks a small Web-based quiz could help meet his objective. However, John Doe only knows JavaScript.

Using the Bean Scripting Framework to help leverage his JavaScript skills, John Doe creates two JSP files, `multiplication_test.jsp` and `multiplication_scoring.jsp`.

In the `multiplication_test.jsp` file, John Doe uses both client-side and server-side JavaScript code to generate a test of 100 random multiplication questions, displayed using a three minute timer. He then writes the `multiplication_scoring.jsp` file to read the data submitted by the `multiplication_test.jsp` file and to generate the scoring results.

John Doe creates the following two files:

multiplication_test.jsp:

```
<html>
<head>
<title>Multiplication Practice Test</title>
<script language="javascript">
var countMin=3;
var countSec=0;
function updateDisplay (min, sec) {
    var disp;
    if (min <= 9) disp = " 0";
    else disp = " ";
    disp += (min + ":");
    if (sec <= 9) disp += ("0" + sec);
    else disp += sec;
    return(disp);
}
```

```
function countDown() {
    countSec--;
    if (countSec == -1) {
        countSec = 59;
        countMin--;
    }
    document.multtest.counter.value = updateDisplay(countMin, countSec);
    if((countMin == 0) &&(countSec == 0)) document.multtest.submit();
    else var down = setTimeout("countDown();", 1000);
}
</script>
</head>
<body bgcolor="#ffffff" onLoad="countDown();">
<%@ page language="javascript" %>
<h1>Three Minute Multiplication Drill</h1>
<hr>
<h2>Remember: this is an opportunity to excel!</h2>
<p>
<form method="POST" name="multtest" action="multiplication_scoring.jsp">
<div align="center">
<table>
<tr>
<td>
<h3>Time left:
<input type="text" name="counter" size="9" value="03:00" readonly>
</h3>
</td>
<td>
<input type="submit" value="Submit for scoring!">
</td>
</tr>
</table>
<table border="1">
<%
var newrow = 0;
var q_num = 0;
function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");
    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");
    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;
    q_num++;
}
for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);
    addQuestion(rand1, rand2);
}
%>
</table>
</div>
</form>
</body>
</html>
```

multiplication_scoring.jsp:

```
<html>
<head>
<title>Multiplication Practice Test Results</title>
</head>
```

```
<body bgcolor="#ffffff">
<%@ page language="javascript" %>
<h1>Multiplication Drill Score</h1>
<hr>
<div align="center">
<table border="1">
<tr><th>Problem</th><th>Correct Answer</th><th>Your Answer</th></tr>
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");
   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}
%>
</table>
<h2>Total Score: <%= total_score %></h2>
<h3><a href="multiplication_test.jsp">Try again?</a></h3>
</div>
</body>
</html>
```

Follow these steps to see how John Doe uses BSF to implement JavaScript in a JSP application:

1. Give your files a `.jsp` extension.
2. Use server-side JavaScript code in your application.

   The `multiplication_test.jsp` file incorporates both client-side and server-side JavaScript. Server-side JavaScript is similar to client-side JavaScript; the primary difference consists of using a different set of objects. Whereas client-side Javascript programmers invoke document and window objects, server-side JavaScript programmers, using the Bean Scripting Framework, invoke a set of objects provided by the JSP technology. Also, client-side scripts are enclosed in <script> tags, but server-side scripts use JSP scriptlet and expression tags.
3. Examine the following blocks of code:

   ```
   <script language="javascript">
   var countMin=3;
   var countSec=0;
   function updateDisplay (min, sec) {
       var disp;
       if (min <= 9) disp = " 0";
       else disp = " ";
       disp += (min + ":");
       if (sec <= 9) disp += ("0" + sec);
   ```

```
    else disp += sec;
    return(disp);
}
function countDown() {
    countSec--;
    if (countSec == -1) {
        countSec = 59;
        countMin--;
    }
    document.multtest.counter.value = updateDisplay(countMin, countSec);
    if((countMin == 0) && (countSec == 0)) document.multtest.submit();
    else var down = setTimeout("countDown();", 1000);
}
</script>
....
<body bgcolor="#ffffff" onLoad="countDown();">
...
<form method="POST" name="multtest" action="multiplication_scoring.jsp">
...
<input type="text" name="counter" size="9" value="03:00" readonly>
...
```

The JavaScript code contained in the <script> block implements a timer set within the <input> field named `counter`. The `onLoad` event handler in the <body> tag causes the browser to load and execute the code when the the page is loaded.

The `document.multtest.submit()` statement causes the form named `multtest` to be submitted when the timer expires.

4. Identify the code to the BSF function.

   The following code example, from the `multiplication_test.jsp` file, displays the use of a JSP directive. This directive tells the WebSphere Application Server BSF function that this file is using the JavaScript language, and that the JavaScript code is enclosed by the `<% ... %>` scriptlet tags. The `out` implicit JSP object in this code example, creates the body section of a table from 100 randomly generated questions.

```
...
<%@ page language="javascript" %>
...
<%
var newrow = 0;
var q_num = 0;

function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");

    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");

    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;

    q_num++;
}

for (var i = 0; i < 100; i++) {
```

```
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);

    addQuestion(rand1, rand2);
}

%>
...
```

5. Read the results.

   To score the results of the practice drill, John Doe uses the `request` implicit JSP object in the `multiplication_scoring.jsp` file to obtain the `POST` data created within the <form> tags in the `multiplication_test.jsp` file.

   The `multiplication_scoring.jsp` file uses the `POST` data to build an output file containing the original question, the student's answer, and the correct answer, and then prints the text in a table format using the `out` implicit object.

   The following code example from the `multiplication_scoring.jsp` file illustrates the use of the `request` and `out` JSP objects:

```
...
<%@ page language="javascript" %>
...
<%
var total_score = 0;
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;
    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");
    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}
var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");
   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}

%>
...
<h2>Total Score: <%= total_score %></h2>
...
```

   Note: Although using separate scriptlet blocks of code for different portions of a conditional expression is common in JSP files implemented in Java, it is invalid for JSP files implemented using JavaScript through the Bean Scripting Framework. The JavaScript code must be entirely contained within the scriptlet tags.

   The following code example illustrates invalid usage:

```
<% if (pass == 0) %>
    <i>pass is true</i>
<% else %>
    <i>pass is not true</i>
```

Deploying the BSF application

You assemble and deploy BSF applications in the same manner as JSP applications. Review the Assembling applications article for more information.

Deploy the BSF code examples in WebSphere Application Server to view this applications processing and output. Use the following quick steps to deploy the application.

The intent of these "quick steps" is to provide you with instant application output. However, the supported method for deployment is the same as for standard JSP files.

1. Use the DefaultApplication to add your BSF files.

   Copy your `.jsp` files to the DefaultApplication directory:

   `<`*`app server install directory`*`>/installedApps/<`*`node name`*`>/DefaultApplication.ear/DefaultApplication.war`
2. Start the application server.
3. Open a browser and request your BSF application.

   Use the following URL to request your application:

   `http://hostName:9080/<`*`jspFileName`*`>.jsp`

Example: Bean Scripting Framework code example

The following code examples show how to implement JavaScript using the Bean Scripting Framework (BSF).

For a quick demonstration of the BSF function, copy these code examples into 2 separate files, and deploy them in WebSphere Application Server using the instructions in the BSF scenario article.

Multiplication practice test

```
<html>
<head>
<title>Multiplication Practice Test</title>
<!--
This file and its companion, multiplication_score.jsp, illustrate the
use of ECMAScript within the BSF framework. The task is a simple
timed math quiz, which is 3 minutes in duration.  When the quiz ends,
the score is computed and displayed.
Users are  then asked if they wish to try
the quiz again.
-->

<!--
This code fragment  displays and  updates the quiz
countdown in client side JavaScript code.
-->
<script language="javascript">
var countMin=3;
var countSec=0;

// This code computes the current countdown time.
function updateDisplay (min, sec) {
```

```
    var disp;

    if (min <= 9) disp = " 0";
    else disp = " ";

    disp += (min + ":");

    if (sec <= 9) disp += ("0" + sec);
    else disp += sec;

    return(disp);
}

//This code fragment displays  the current countdown time in the user's
//browser window,and submits the results for scoring when the countdown
//ends.

function countDown() {
    countSec--;
    if (countSec == -1) {
        countSec = 59;
        countMin--;
    }
    document.multtest.counter.value = updateDisplay(countMin, countSec);
    if((countMin == 0) && (countSec == 0)) document.multtest.submit();
    else var down = setTimeout("countDown();", 1000);
}

</script>
</head>
<body bgcolor="#ffffff" onLoad="countDown();">

<!--
The body of the quiz runs as  JavaServer Pages (JSP) code using BSF.
The code  outputs the problems in table format using the  POST method
and invokes  the scoring module when the user chooses to end the quiz
or when the countdown ends.
-->
<%@ page language="javascript" %>

<h1>Three Minute Multiplication Drill</h1>
<hr>

<h2>Remember: this is an opportunity to excel!</h2>
<p>

<form method="POST" name="multtest" action="multiplication_scoring.jsp">
<div align="center">
<table>
<tr>
<td>
<h3>Time left:
<input type="text" name="counter" size="9" value="03:00" readonly>
</h3>
</td>
<td>
<input type="submit" value="Submit for scoring!">
</td>
</tr>
</table>
<table border="1">
<%
var newrow = 0;
var q_num = 0;

// This code  generates  a new random multiplication problem up to the number
//twelve, and enters  it into the table of problems.
```

```
function addQuestion(num1, num2) {
    if (newrow == 0) out.println("<tr>");

    out.println("<td>");
    out.println(num1 + " x " + num2 + " = ");
    out.println("</td><td>");
    out.print("<input name=\"" + q_num + "|" + num1 + ":" + num2 + "\" ");
    out.println("type=\"text\" size=\"10\">");
    out.println("</td>");

    if (newrow == 3) {
        out.println("</tr>");
        newrow = 0;
    }
    else newrow++;

    q_num++;
}


//This code obtains two random operands and formats 100 quiz problems.

for (var i = 0; i < 100; i++) {
    var rand1 = Math.ceil(Math.random() * 12);
    var rand2 = Math.ceil(Math.random() * 12);

    addQuestion(rand1, rand2);
}

%>
</table>
</div>
</form>

</body>
</html>
```

Multiplication practice test results

```
<html>
<head>
<title>Multiplication Practice Test Results</title>
</head>
<body bgcolor="#ffffff">

<!--
This JSP code is invoked when the user submits a math quiz for scoring,
or when the quiz countdown expires. The JSP code tabulates the problem list,
the correct answer, the user's answer, and scores the test. It then offers
the user an opportunity to try the quiz again.
-->
<%@ page language="javascript" %>

<h1>Multiplication Drill Score</h1>
<hr>

<div align="center">
<table border="1">
<tr><th>Problem</th><th>Correct Answer</th><th>Your Answer</th></tr>
<%
var total_score = 0;

// This code parses the submitted form, extracts the a problem generated by the
// multiplication_test.jsp file, outputs it, computes the correct answer,
// and displays  this information and  the user answer.  The code scores
// the quiz using a running sum of correct answers.
```

```
function score (current, pos1, pos2) {
    var multiplier = current.substring(pos1 + 1, pos2);
    var multiplicand = current.substring(pos2 + 1, current.length());
    var your_product = request.getParameterValues(current)[0];
    var true_product = multiplier * multiplicand;

    out.println("<tr>");
    out.println("<td>" + multiplier + " x " + multiplicand + " = </td>");
    out.println("<td>" + true_product + "</td>");

    if (your_product == true_product) {
        total_score++;
        out.print("<td bgcolor=\"\#00ff00\">");
    }
    else {
        out.print("<td bgcolor=\"\#ff0000\">");
    }
    out.println(your_product + "</td>");
    out.println("</tr>");
}


// This is the main body of the scoring application. It parses the posted quiz,
// and calls  the score() function to score remaining problems.

var equations = request.getParameterNames();
while(equations.hasMoreElements()) {
   var currElt = equations.nextElement();
   var splitPos1 = currElt.indexOf("|");
   var splitPos2 = currElt.indexOf(":");

   if (splitPos1 >=0 && splitPos2 >= 0) score(currElt, splitPos1, splitPos2);
}

%>
</table>

<h2>Total Score: <%= total_score %></h2>
<h3><a href="/multiplication_test.jsp">Try again?</a></h3>
</div>

</body>
</html>
```

Web modules

A Web module represents a Web application. A Web module is created by assembling servlets, JavaServer Pages (JSP) files, and static content such as HyperText Markup Language (HTML) pages into a single deployable unit. Web modules are stored in Web archive (WAR) files, which are standard Java archive files.

A Web module contains:

- One or more servlets, JSP files, and HTML files.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file.

  The file, named `web.xml`, declares the contents of the module. It contains information about the structure and external dependencies of Web components in the module and describes how the components are used at run time.

You can create Web modules as stand-alone applications, or you can combine Web modules with other modules to create J2EE applications. You install and run a Web module in the Web container of an application server.

Assembling Web applications

Assemble a Web module to contain servlets, JavaServer page (JSP) files, and related code artifacts. (Group enterprise beans, client code, and resource adapter code in separate modules). After assembling a Web module, you can install it as a stand-alone application or combine it with other modules into an enterprise application.

Use the Assembly Toolkit to assemble a Web module in any of the following ways:
- Import an existing Web module (WAR file).
- Create a new Web module.
- Copy code artifacts (such as servlets) from one Web module into a new Web module.

Although you can input various properties for Web archives, available properties are specific to the Servlet, JSP, and J2EE specification level.

1. Start the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
4. Migrate WAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import your WAR files to the Assembly Toolkit.
5. Create a new Web module.
6. Copy code artifacts (such as servlets) from one Web module into a new Web module.
7. Verify the contents of the new Web module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Web Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the Web module in a Navigator view.

Using the AAT to assemble Web modules

If you want to use existing Java 2 Enterprise Edition (J2EE) 1.2 Web modules in your J2EE 1.3 application, migrate them to J2EE 1.3 first.

Assemble a Web module to contain servlets, JSP files, and related code artifacts. (Group enterprise beans, client code, and resource adapter code in separate modules).

A Web module can be installed as a stand-alone application or can be combined with other modules into an enterprise application.

The Application Assembly Tool (AAT) provides flexibility in assembling Web modules. Options described below include:
- Importing an existing Web module (WAR file)
- Creating a new Web module

- Copying code artifacts (such as servlets) from one Web module into a new Web module

Although you can input various properties for Web archives, available properties are specific to the Servlet, JSP, and J2EE specification level.

1. Start the AAT.
2. From the New tab, select **Web Module**. Click **OK**. The navigation tree now displays various sets of properties for configuring the new Web module.
3. Use the property dialog shown in the AAT workspace to change the default file name and location.
   a. It is recommended that you change the display name so that it differs from the file name.
   b. If you like, change the temporary location of the Web module from the default location, install_root/bin.
4. Add at least one Web component (servlet or JSP file) to the module. You must add at least one Web component, using one of the following methods.
   - Import an existing WAR file containing Web components.
     a. In the navigation tree, right-click the **Web Components** folder.
     b. Select **Import** from its right-click menu.
     c. Use the file browser to locate and select the archive file for the module.
     d. Click **Open**. The Web applications in the selected archive are displayed.
     e. Select a Web application. Its Web components are displayed in the workspace.
     f. Select the servlets or JSP files to be added and click **Add**. The components are displayed in the **Selected Components** window.
     g. Click **OK**. The properties associated with the archive are also imported. The property dialog boxes in the workspace are populated automatically with values.
     h. Double-click the Web Components icon to verify that the servlets or JSP files are included in the module.
     i. Double-click the **Web Components** icon to verify that the servlets or JSP files are included in the module.
     j. Save the Web module.
   - Copy and paste archive files from an existing module into the new Web component.
   - Create a new Web component.
     a. In the navigation tree, right-click the **Web Components** folder.
     b. Select **New** from its right-click menu.
     c. When the new module is displayed, enter a component name and choose a component type.
     d. Use the file browser to locate and select the archive file for the module.
     e. Click the plus sign (+) to verify its contents and enter assembly properties.
     f. In the **New Web Component** property dialog box, click **OK**.
     g. Verify that the Web component has been added to the module by double-clicking the **Web components** icon in the navigation tree.
     h. Click the component to view its corresponding property dialog box in the bottom portion of the pane.
5. Enter assembly properties for each Web component.
   a. Click the plus sign (+) next to the each component to reveal its property groups.
   b. Right-click each property group icon and click **New** to display properties in the workspace.

6. Specify additional properties for the Web module.

   Right-click each property group's icon. Choose **New** to add new values, or edit existing values in the property pane. (Click **Help** for descriptions of the settings).

   Note that if you add a security constraint, you must add at least one Web resource collection.
7. Add any other files needed by the application.
   - In the navigation tree, click the plus sign (+) next to the Files icon. Right-click **Add Class Files**, **Add JAR Files**, or **Add Resource Files**. Select **Add Files**.
   - Add files, using the Add Files dialog.
8. Save the application.

Assemble other new modules of your choice, if needed:
- Assembling EJB modules.
- Assembling application client modules.
- Assembling resource adapter modules.

You can also migrate existing modules.

Another option is to proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

Context parameters

A servlet context defines a server's view of the Web application within which the servlet is running. The context also allows a servlet to access resources available to it.

Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use. These properties declare a Web application's parameters for its context. They convey setup information, such as a webmaster's e-mail address or the name of a system that holds critical data.

Security constraints

Security constraints determine how Web content is to be protected.

These properties associate security constraints with one or more Web resource collections. A constraint consists of a Web resource collection, an authorization constraint and a user data constraint.
- A Web resource collection is a set of resources (URL patterns) and HTTP methods on those resources. All requests that contain a request path that matches the URL pattern described in the Web resource collection is subject to the constraint. If no HTTP methods are specified, then the security constraint applies to all HTTP methods.
- An authorization constraint is a set of roles that users must be granted in order to access the resources described by the Web resource collection. If a user who requests access to a specified URI is not granted at least one of the roles specified in the authorization constraint, the user is denied access to that resource.
- A user data constraint indicates that the transport layer of the client or server communications process must satisfy the requirement of either guaranteeing content integrity (preventing tampering in transit) or guaranteeing confidentiality (preventing reading while in transit).

Servlet mappings

A servlet mapping is a correspondence between a client request and a servlet.

Servlet containers use URL paths to map client requests to servlets, and follow the URL path-mapping rules as specified in the Java Servlet specification. The container uses the URI from the request, minus the context path, as the path to map to a servlet. The container chooses the longest matching available context path from the list of Web applications that it hosts.

Invoker attributes

Invoker attributes are used by the servlet that implements the invocation behavior.

Error pages

Error page locations allow a servlet to find and serve a URI to a client based on a specified error status code or exception type.

These properties are used if the error handler is another servlet or JSP file. The properties specify a mapping between an error code or exception type and the path of a resource in the Web application. The container examines the list in the order that it is defined, and attempts to match the error condition by status code or by exception class. On the first successful match of the error condition, the container serves back the resource defined in the Location property.

File serving

File serving allows a Web application to serve static file types, such as HTML. File-serving attributes are used by the servlet that implements file-serving behavior.

Initialization parameters

Initialization parameters are sent to a servlet in its HttpConfig object when the servlet is first started.

Servlet caching

Dynamic caching can be used to improve the performance of servlet and JavaServer Pages (JSP) files by serving requests from an in-memory cache. Cache entries contain the servlet’s output, results of the servlet’s execution, and metadata.

Web components

A web component is a servlet, Java Server Page (JSP), or HTML file. One or more web components make up a web module.

Web property extensions

Web property extensions are IBM extensions to the standard deployment descriptors for Web applications. These extensions include mime filtering and servlet caching.

Web resource collections

A Web resource collection defines a set of URL patterns (resources) and HTTP methods belonging to the resource.

HTTP methods handle HTTP-based requests, such as GET, POST, PUT, and DELETE. A URL pattern is a partial Uniform Resource Locator that acts as a template for matching the pattern with existing full URLs in an attempt to find a valid file.

Welcome files

A Welcome file is an entry point file (for example, `index.html`) for a group of related HTML files.

Welcome files are located by using a group of partial URIs. The Web container uses the partial URIs to find a valid file when the initial URI is not found.

Context parameter assembly settings

A servlet context defines the server view of the Web application within which the servlet is running. The context also allows a servlet to access resources available to it. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

Parameter name (Required, String)

Specifies the name of a parameter, for example, dataSourceName.

Data type String

Parameter value (Required, String)

Specifies the value of a parameter, for example, `jdbc/sample`.

Data type String

Description

Contains a description of the context parameter.

Data type String

Initialization parameter assembly settings

Use this page to specify the initialization parameters that are sent to a servlet in its HttpConfig object when the servlet is first started.

To view this page in the Application Assembly Tool, click

Web Modules > *component_instance* > **Web Components** > *component_instance* > **Initialization Parameters**

Parameter name (Required, String)

Specifies the name of an initialization parameter.

Data type String

Parameter value (Required, String)

Specifies the value of the initialization parameter.

Data type String

Description

Contains text describing the use of the parameter.

Data type String

Filter assembly settings

Use the Filter panel to configure your filter settings.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

Filter name

Specifies the logical name of the filter. This name maps the filter.

Data type	String

Class

Specifies the fully qualified classname of the filter.

Data type	String

Description

Provides a description of the filter.

Data type	String

JavaServer Pages attribute assembly settings

Use the JavaServer Pages (JSP) attributes page to set JSP attributes that are used by servlets that implement JSP processing behavior.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Assembly Property Extensions**

5.0.2 Or use the Chapter 17, "Assembling applications with the Assembly Toolkit," on page 879 to set the JavaServer Pages attribute assembly settings.

JSP Attribute (Name)

Specifies the name of an attribute.

Data type	String

JSP Attribute (Value)

Specifies the value of an attribute.

Data type	String

Supported JSP attributes

The WebSphere JSP container supports the following JSP attributes:

classdebuginfo

Indicates the compiler should include debugging information in the generated classfile.

classdebuginfo	true or false

Default is false.

classpath

Specifies an additional classpath for compiling the generated servlets.

classpath	*classpath* or *null*

Default is *null*.

deprecation

Indicates the compiler should generate deprecation warnings when compiling the generated Java source.

deprecation	`true` or `false`

Default is `false`.

disableJspRuntimeCompilation

Indicates the runtime behavior of the JSP compiler. If this option is set to true, the JSP compiler does not compile or translate the JSP files, and the JSP engine only loads precompiled classfiles.

disableJspRuntimeCompilation	`true` or `false`

Default is `false`.

ieClassID

Indicates the Java plugin COM class ID for Internet Explorer. The <jsp:plugin> tags use this value.

ieClassID	*classid*

Default is `clsid:8AD9C840-044E-11D1-B3E9-00805F499D93`.

javaEncoding

Indicates the Java platform encoding to use to generate the JSP page servlet.

javaEncoding	*encoding value*

Default is `UTF-8`.

jspCompilerPath

Indicates the path of the compiler to use for compiling JSP pages.

jspCompilerPath	*path name* or *null*

Default is *null*.

keepgenerated

Indicates the Java files generated by the JSP compiler during the translation phase of the processing should be kept.

keepgenerated	`true` or `false`

Default is `false`.

largefile

Specifies support for large files. When the Java code is generated, the HTML data in a JSP file is stored separately instead of being saved as constant string data in the generated servlet.

largefile	true or false

Default is false.

mappedfile

Indicates the compiler should generate Java source that includes a print statement for every line in the JSP file. Use this option for debugging purposes only. It is not recommended for production environments because the mappedfile option generates too many out.print() statements.

mappedfile	true or false

Default is false.

scratchdir

Specifies the directory where the generated classfiles are created.

scratchdir	*directory name.*

Default is [WAS_INSTALL_ROOT]/temp.

Note: The system property com.ibm.websphere.servlet.temp.dir can be used to set the scratchdir option on a server-wide basis. This setting, if it is present, overrides the system property.

usePageTagPool

Enables or disables the reuse of custom tag handlers on an individual JavaServer Page basis.

usePageTagPool	true or false

Default is false.

Note: Enabling custom tag handler reuse might reveal problems in your tag handler code regarding the tags ability to be reused. A custom tag handler should always do two things:

1. The release() method of the tag handler should reset its state and release any private resources that it might have used. The JSP engine guarantees the release() method will be called before the tag handler is garbage collected.
2. In the doEndTag() method, all instance states associated with this instance must be reset.

useThreadTagPool

Enables or disables the reuse of custom tag handlers on a per request thread basis.

useThreadTagPool	true or false

Default is false.

The note in the `usePageTagPool` attribute description also applies to the`useThreadTagPool` attribute.

verbose

Indicates the compiler should generate verbose output when compiling the generated Java source code.

verbose	`true` or `false`

Default is `false`.

Multipurpose Internet Mail Extensions (MIME) filter assembly settings

Use this page to configure Multipurpose Internet Mail Extensions (MIME) filters.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Assembly Property Extensions**

Component name (Required, String)

Specifies the name of the servlet or JavaServer Pages(TM) (JSP) file. This name must be unique within the Web module.

Data type	String

Display name

Specifies a short name that is intended for display by GUIs.

Data type	String

Description

Contains a description of the servlet or JSP file.

Data type	String

Component type

Specifies the type of Web component. Valid values are `servlet` or `JSP` file.

Data type	String

Class name (Required, String)

Specifies the full path name for the servlet class.

Data type	String

JSP file (Required, String)

Specifies the full path name for the JSP file.

Data type	String

Load on startup

Indicates whether this servlet loads at the startup of the Web application.
The default is `false` (the check box is not selected). Also specifies a positive integer indicating the order in which to load the servlet. Lower integers are loaded before

higher integers. If no value is specified, or if the value specified is not a positive integer, the container is free to load the servlet at any time in the startup sequence.

Data type	String

Small icon

Specifies a JPEG or GIF file containing a small image (16x16 pixels). Use the image as an icon to represent the Web component in a GUI.

Data type	JPEG, GIF

Large icon

Specifies a JPEG or GIF file containing a large image (32x32 pixels). Use the image as an icon to represent the Web component in a GUI.

Data type	JPEG, GIF

Page list assembly settings

Page lists allow you to avoid hardcoding URLs in servlets and JSP files.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Web Components**

Name

Specifies the name of the markup language--for example, Hypertext Markup Language (HTML), Wireless Markup Language (WML), and Voice Extensible Markup Language (VXML).

Data type	String

MIME Type

Specifies the Multi-Purpose Internet Mail Extensions (MIME) type of the markup language, for example, text/html and text/x-vxml.

Data type	String

Error Page

Specifies the name of an error page.

Data type	String

Default Page

Specifies the name of a default page.

Data type	String

Pages - Name

Specifies the name of the page to serve, for example, *StockQuoteRequest.page*.

Data type	String

Pages - URI

Specifies the URI of the page to serve, for example, `examples/StockQuoteHTMLRequest.jsp`.

Data type	String

Security constraint assembly settings

Use the Security constraints panel to configure security constraints.

To view this Application Assembly Tool (AAT) panel, open an existing or create a new Web module. Right-click **Security Constraints** from the left navigation menu. Click **New**.

If multiple security constraints are specified, the container uses the "first match wins" rule when processing a request to determine what authentication method to use, or what authorization to allow

Security constraint name

Specifies the name of the security constraint.

Data type	String

Authorization Constraints - Roles

Specifies the user roles that are permitted access to this resource collection.

Data type	String

Authorization Constraints - Description

Contains a description of the authorization constraints

Data type	String

User Data Constraints - Transport guarantee

Indicates how data communicated between the client and the server is to be protected.

Specifies that the protection for communications between the client and server is None, Integral, or Confidential.

- None means that the application does not require any transport guarantees.
- Integral means that the application requires that the data sent between the client and the server must be sent in such a way that it cannot be changed in transit.
- Confidential means that the application requires that the data must be transmitted in a way that prevents other entities from observing the contents of the transmission.

In most cases, Integral or Confidential indicates that the use of SSL is required.

Data type	String

User Data Constraints - Description

Contains a description of the user data constraints.

Data type	String

Servlet mapping assembly settings

A servlet mapping is a correspondence between a client request and a servlet.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

URL pattern (Required, String)

Specifies the URL pattern of the mapping.

The URL pattern must conform to the Servlet specification. Use the following syntax:

- A string beginning with a slash character (/) and ending with the slash and asterisk characters (/*) represents a path mapping.
- A string beginning with the characters *. represents an extension mapping.
- All other strings are used as exact matches only.
- A string containing only the slash character (/) indicates that the servlet specified by the mapping becomes the default servlet of the application. In this case, the servlet path is the request Uniform Resource Identifier (URI) minus the context path, and the path information is null.

Data type	String

Servlet (Required, String)

Specifies the name of the servlet associated with the URL pattern.

Data type	String

Tag library assembly settings

Use this page to define the tag library parameters.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

Java ServerPages (JSP) tag libraries contain classes for common tasks such as processing forms and accessing databases from JSP files.

Tag library file name (Required, String)

Specifies a file name relative to the location of the `web.xml` document, identifying a tag library used in the Web application.

Data type	String

Tag library location (Required, String)

Contains the location, as a resource relative to the root of the Web application, where you can find the Tag Library Definition file for the tag library.

Data type	String

Welcome file assembly settings

Use this page to configure your welcome page.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

Welcome file (Required, String)

The Welcome file list is an ordered list of partial URLs with no trailing or leading slash characters (/).
The Web server appends each file in the order specified and checks whether a resource in the Web archive (WAR) file is mapped to that request Uniform Resource Identifier (URI). The container forwards the request to the first resource in the WAR file that matches.

Data type	String

Servlet caching configuration assembly settings

Use this page to configure your cache groups. Access this page by traversing the following path in the Application AssemblyTool: **Web Modules** > *component_instance* > **Assembly Property Extensions**

5.0.2 Or use the Assembly Toolkit to set the servlet caching settings.

The properties on the General tab define a cache group and govern how long an entry remains in the cache. The properties on the ID Generation tab define how cache IDs are built and the criteria used to cache or invalidate entries. The properties on the Advanced tab define external cache groups and specify custom interfaces for handling servlet caching.

Caching group name

Specifies a name for the group of servlets or JavaServer Pages (JSP) files to cache.

Priority

Defines the default priority for cached servlets. Specify as an integer. The default value is 1.
Priority is an extension of the Least Recently Used (LRU) caching algorithm. It represents the number of cycles through the LRU algorithm that an entry is guaranteed to stay in the cache. The priority represents the length of time that an entry remains in the cache before becoming eligible for removal. On each cycle of the algorithm, the priority of an entry is decremented. When the priority reaches zero, the entry is eligible for invalidation. If an entry is requested while in the cache, its priority is reset to the priority value. Regardless of the priority value and the number of requests, an entry is invalidated when its timeout occurs. Consider increasing the priority of a servlet or JSP file when it is difficult to calculate the output of the servlet or JSP file or when the servlet or JSP file is executed more often than average. Priority values should be low. Higher values do not yield much improvement but use extra LRU cycles. Use timeout to guarantee the validity of an entry. Use priority to rank the relative importance of one entry to other entries. Giving all entries equal priority results in a standard LRU cache that increases performance significantly.

Timeout

Specifies the length of time, in seconds, that a created cache entry remains in the cache.
When this time elapses, the entry is removed from the cache. If the timeout is zero or a negative number, the entry does not time out. It is removed when the cache is full or programmatically, from within an application.

Invalidate only

Specifies that invalidations for a servlet take place, but that no servlet caching is performed.

For example, you can use this property to prevent caching of control servlets. Control servlets treat HTTP requests as commands and execute those commands. By default, this check box is not selected.

Caching group members
Specifies the names of the servlets or JSP files to cache. The URIs are determined from the servlet mappings.

Use URIs for cache ID building
Specifies whether or not to use the URI of the requested servlet to create a cache ID. By default, URIs are used.

Use specified string
Specifies a string representing a combination of request and session variables to use for creating cache IDs. This property defines request and session variables, and the cache uses the values of these variables to create IDs for the entries.

Variables - ID
Specifies the name of a request parameter, request attribute, session parameter, or cookie.

Variables - Type
Specifies the type of variable inidcated in the ID field. The valid values are Request parameter, Request attribute, Session parameter, or Cookie.

Variables - Method
Specifies the name of a method in the request attribute or session parameter. The output of this method is used to generate cache entry IDs. If this value is not specified, the toString method is used by default.

Variables - Data ID
Specifies a string that, combined with the value of the variable, generates a group name for the cache entry. The cache entry is placed in this group. You can invalidate this group.

Variables - Invalidate ID
Specifies a string that is combined with the value of the variable on the request or session to form a group name. The cache invalidates the group name.

Required
Specifies whether a value must exist in the request. If this check box is selected, and either the request parameter, request attribute, session parameter, or the method is not specified, the request is not cached.

External cache groups - Group name
Specifies the name of the external cache group to which this servlet is published.

ID generator
Specifies a user-written interface for handling parameters, attributes, and sessions. The value must represent a full package and class name of a class extending com.ibm.websphere.servlet.cache.IdGenerator. The properties specified in the Application Assembly Tool are used and passed to the IdGenerator in the initialize method inside a com.ibm.websphere.servlet.cache.CacheConfig object.

Data type	String

Meta data generator

Specifies a user-written interface for handling invalidation, priority levels, and external cache groups.

The value must represent the full package and class name of a class extending com.ibm.websphere.servlet.cache.MetaDataGenerator. The properties specified in the Application Assembly Tool are used and passed to the MetaDataGenerator in the initialize method inside a com.ibm.websphere.servlet.cache.CacheConfig object.

Data type	String

Web components assembly settings

Use this page to set the assembly properties for the components that make up a Web module.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Web Components**

5.0.2 Or use the Assembly Toolkit to set the web components assembly settings.

Component name

Specifies the name of the servlet or JavaServer Pages(TM) (JSP) file. This name must be unique within the Web module.

Data type	String

Display name

Specifies a short name that is intended for display by GUIs.

Data type	String

Description

Contains a description of the servlet or JSP file.

Data type	String

Component type

Specifies the type of Web component. Valid values are servlet or JSP file.

Data type	String

Class name

Specifies the full path name for the servlet class.

Data type	String

JSP file

Specifies the full path name for the JSP file.

Data type	String

Load on startup

Indicates whether this servlet loads at the startup of the Web application. The default is false (the check box is not selected).

This field also specifies a positive integer indicating the order in which the servlet is to load. Lower integers are loaded before higher integers. If no value is specified, or if the value specified is not a positive integer, the container is free to load the servlet at any time in the startup sequence.

Data type	Boolean
Default	False

Small icon

Specifies a JPEG or GIF file containing a small image (16x16 pixels). Use the image as an icon to represent the Web component in a GUI.

Data type	String

Large icon

Specifies a JPEG or GIF file containing a large image (32x32 pixels). Use the image as an icon to represent the Web component in a GUI.

Data type	String

Run as role name

Enter a role name that represents the user account under which the servlet executes. The default role name is blank, which indicates the servlet runs under the user that logged into the application server. The role name of ″all role″ indicates the servlet can execute under different users.

Data type	String

Description

In this optional field, enter a description that explains the importance of the role, and where and how the role can be used.

Data type	String

Run as role mode

Indicates a security role that is defined in the enterprise application.

Data type	String

Local Transactions - Unresolved action

Specifies the action the Web container must take if resources in a local transaction are uncommitted by an application. This property is an IBM extension to the standard J2EE deployment descriptor. A local transaction context is provided by the container in the absence of a global transaction context.

Data type	String	
Default	Rollback	
Range valid values are	Commit	Rollback

Additional information about these settings follows:

Commit
: At end of the local transaction context, the container instructs all the unresolved local transactions to commit.

Rollback (default)
: At end of the local transaction context, the container instructs all the unresolved local transactions to rollback.

Web modules assembly settings

Use this page to set the assembly properties for web modules. Web modules are composed of one or more web components.

Access this page by traversing the following path in the Application Assembly Tool:

application_instance > **Web Modules**

File name

Specifies the file name of the Web module, relative to the top level of the application package.

Alternative DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module's JAR file.

This file is the post-assembly version of the deployment descriptor file. (The original deployment descriptor file can be edited to resolve dependencies and security information. Directing the use of the alternative deployment descriptor allows you to keep the original deployment descriptor file intact). The value of the Alternative DD property must be the full path name of the deployment descriptor file relative to the module's root directory. By convention, the file is in the ALT-INF directory. If this property is not specified, the deployment descriptor file is read directly from the module's JAR file.

Context root

Specifies the context root of the Web application. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet.

For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is http://host:port/gettingstarted/MySession.

Classpath

Specifies the class path for resources used by the Web application, relative to the ear file..

If your Web application requires access to classes within an ear file, specify the relative path of the classes in this field.

Display name

Specifies a short name that is intended to be displayed by GUIs.

Description

Contains a description of the Web module.

Distributable

Specifies that this Web application is programmed appropriately to deploy into a distributed servlet container.

Small icon

Specifies a JPEG or GIF file containing a small image (16x16 pixels). The image is used as an icon to represent the module in a GUI.

Large icon

Specifies a JPEG or GIF file containing a large image (32x32 pixels). The image is used as an icon to represent the module in a GUI.

Session configuration

Indicates that session configuration information is present. Checking this box makes the Session timeout property editable.

Session timeout

Specifies a time period, in seconds, after which a client is considered inactive. The default value is zero, indicating that the session timeout never expires.

Login configuration -- Authentication method

Specifies an authentication method to use. As a prerequisite to gaining access to any Web resources protected by an authorization constraint, a user must authenticate by using the configured mechanism.
A Web application can authenticate a user to a Web server by using one of the following mechanisms: HTTP basic authentication, HTTP digest authentication, HTTPS client authentication, and form-based authentication.

- HTTP basic authentication is not a secure protocol because the user password is transmitted with a simple Base64 encoding and the target server is not authenticated. In basic authentication, the Web server requests a Web client to authenticate the user and passes a string called the realm of the request in which the user is to be authenticated.
- HTTP digest authentication transmits the password in encrypted form.
- HTTPS client authentication uses HTTPS (HTTP over SSL) and requires the user to possess a public key certificate.
- Form-based authentication allows the developer to control the appearance of login screens.

The Login configuration properties are used to configure the authentication method that should be used, the realm name that should be used for HTTP basic authentication, and the attributes that are needed by the form-based login mechanism. Valid values for this property are Unspecified, Basic, Digest, Form, and Client certification.

HTTP digest authentication is not supported as a login configuration in this product. Also, not all login configurations are supported in all of the product's global security authentication mechanisms (Local Operating system, LTPA, and custom pluggable user registry). HTTP basic authentication and form-based login authentication are the only authentication methods supported by the Local Operating system user registry. LTPA and the custom pluggable user registry are capable of supporting HTTP basic authentication, form-based login, and HTTPS client authentication.

Login configuration -- Realm name

Specifies the realm name to use in HTTP basic authorization. It is based on a user name and password, sent as a string (with a simple Base64 encoding).
An HTTP realm is a string that allows URIs to be grouped together. For example, if a user accesses a secured resource on a Web server within the "finance realm," subsequent access to the same or different resource within the same realm does not result in a repeat prompt for a user ID and password.

Login configuration -- Login page

Specifies the location of the login form. If form-based authentication is not used, this property is disabled.

Form Login Config -- Error page
Specifies the location of the error page. If form-based authentication is not used, this property is disabled.

Reload interval
Specifies a time interval, in seconds, in which the file system of the Web application is scanned for updated files. The default is 3 seconds.

Reloading enabled
Specifies whether file reloading is enabled. The default is true.

Default error page
Specifies a file name for the default error page. If no other error page is specified in the application, this error page is used.

Additional classpath
Specifies the full class path that will be used to reference classes outside of those specified in the archive.
If your Web application requires access to classes not contained in the archive file, specify the full path for those classes in this field.

File serving enabled
Specifies whether file serving is enabled. File serving allows the application to serve static file types, such as HTML and GIF. File serving can be disabled if the application contains only dynamic components. The default value is true.

Directory browsing enabled
Specifies whether directory browsing is enabled. Directory browsing allows the application to browse disk directories. Directory browsing can be disabled if, you want to protect data. The default value is true.

Serve servlets by classname
Specifies whether a servlet can be served by requesting its class name.
Usually, servlets are served only through a URI reference. The class name is the actual name of the servlet on disk. For example, a file named SnoopServlet.java compiles into SnoopServlet.class. (This is the class name.) SnoopServlet.class is normally invoked by specifying snoop in the URI. However, if the *Serve servlets by classname* property is enabled, the servlet is invoked by specifying SnoopServlet. The default value is true.

Virtual hostname
Specifies a virtual host name. A virtual host is a configuration enabling a single host machine to resemble multiple host machines. This property allows you to bind the application to a virtual host in order to enable execution on that virtual host.

Filter mappings
Specifies the filter mapping declarations in this application. The container uses the filter mapping declarations to decide on the type and order of filters to apply to a request.
After the container matches the request URI to a servlet, for each filter mapping element, it determines what filters to apply based on the servlet name or the URL pattern, depending on the style specified. Filters are invoked in the same order as the one specified in the list of filter mapping elements. The value that you specify for the filter name must be the same value as that specified in the <filter><filtername> sub-element declarations in the deployment descriptor.

Assembly property extensions

Use this panel to configure WebSphere Application Server specific Web module extensions, or also referred to as assembly property extensions.

This panel lists the extensions that can be configured through the tool.

Reach the applicable extension panel by clicking on the panel name in the navigation at the left, or by double-clicking the attribute name in the list provided.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Assembly Property Extensions**

Assembly properties for a Web module include:
- File serving attributes
- Invoker attributes
- JavaServer Pages (JSP) attributes
- Multipurpose Internet Mail Extensions (MIME) filters
- Servlet caching configurations

File serving attribute assembly settings

File serving allows a Web application to serve static file types, such as HTML. File-serving attributes are used by the servlet that implements file-serving behavior.

Access this page by traversing the following path in the Application AssemblyTool:

Web Modules > *component_instance* > **Assembly Property Extensions**

File Serving Attribute (Name)

Specifies the name of an attribute.

Data type	String

File Serving Attribute (Value)

Specifies the value of an attribute.

Data type	String

Invoker attribute assembly settings

Invoker attributes are used by the servlet that implements the invocation behavior.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance* > **Assembly Property Extensions**

Invoker Attribute (Name)

Specifies the name of an attribute.

Data type	String

Invoker Attribute (Value)

Specifies the value of an attribute.

Data type	String

Error page assembly settings

Error page locations allow a servlet to find and serve a URI to a client based on a specified error status code or exception type. These properties are used if the error handler is another servlet or JSP file.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

The error page properties specify a mapping between an error code or exception type and the path of a resource in the Web application. The container examines the list in the order that it is defined, and attempts to match the error condition by status code or by exception class. On the first successful match of the error condition, the container serves back the resource defined in the Location property.

Error code

Indicates that the error condition is a status code.

Data type	Integer

Error Code (Required, String)

Specifies an HTTP error code, for example, 404.

Data type	String

Exception

Indicates that the error condition is an exception type.

Data type	String

Exception type name (Required, String)

Specifies an exception type.

Data type	String

Location (Required, String)

Contains the location of the error-handling resource in the Web application.

Data type	String

Web resource collections security constraint properties

A Web resource collection defines a set of URL patterns or resources and HTTP methods belonging to the resource, which define the security constraints for a Web component.

Access this page by traversing the following path in the Application Assembly Tool:

Web Modules > *component_instance*

HTTP methods handle HTTP-based requests, such as GET, POST, PUT, and DELETE. A URL pattern is a partial Uniform Resource Locator that acts as a template for matching the pattern with existing full URLs in an attempt to find a valid file.

Web resource name

Specifies the name of a Web resource collection.

Data type	String

Web resource description

Contains a description of the Web resource collection.

HTTP methods

Specifies the HTTP methods to which the security constraints apply. If no HTTP methods are specified, then the security constraint applies to all HTTP methods. The valid values are GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE.

Data type	String

URL pattern

Specifies URL patterns for resources in a Web application. All requests that contain a request path that matches the URL pattern are subject to the security constraint.

Data type	String

Troubleshooting tips for Web application deployment

Deployment of a Web application is successful if you can access the application by typing a Uniform Resource Locator (URL) in a browser, or if you can access the application by following a link.

If you cannot access your application, follow these steps to eliminate some common errors that can occur during migration or deployment.

Web module does not run in WebSphere Application Server Version 5.

Symptom	Your Web module does not run when you migrate it to Version 5
Problem	In Version 4.x, the classpath setting that affected visibility was *Module Visibility Mode*. In Version 5, you must use class loader policies to set visibility.
Recommended response	Reassemble an existing module, or change the visibility settings in the class loader policies. in the class loader policies. See article Migration of module visibility modes from Version 4.x for more information and examples.

Welcome page is not visible.

Symptom	You cannot access an application with a Web path of: `/webapp/myapp`
Problem	The default welcome page for a Web application is assumed to be *index.html*. You cannot access the default page of the *myapp* application unless it is named *index.html*.
Recommended response	To identify a different welcome page, modify the properties of the Web module during assembly. See article Assembling Web modules for more information.

HTML files are not found.

Symptom
: Your Web application ran successfully on prior versions, but now you encounter errors that the welcome page (typically *index.html*), or referenced HTML files are not found:

```
Error 404: File not found: Banner.html
Error 404: File not found: HomeContent.html
```

Problem
: For security and consistency reasons, Web application URLs are now case-sensitive on all operating systems.

Suppose the content of the index page is as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 5.0 Frameset//EN">
<HTML>
<TITLE>
Insurance Home Page
</TITLE>
 <frameset rows="18,80">
 <frame src="Banner.html" name="BannerFrame" SCROLLING=NO>
 <frame src="HomeContent.html" name="HomeContentFrame">
 </frameset>
</HTML>
```

However the actual file names in the `\WebSphere\AppServer\installedApps\...` directory where the application is deployed are:

```
banner.html
homecontent.html
```

Recommended response
: To correct this problem, modify the *index.html* file to change the names *Banner.html* and *HomeContent.html* to *banner.html* and *homecontent.html* to match the names of the files in the deployed application.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

Modifying the default Web container configuration

The Web container is created initially with default properties values suitable for simple Web applications. However, these values might not be appropriate for more complex Web applications.

Your application is considered complex if it requires any of the following features:
- virtual host
- servlet caching
- special client request loads
- persistent HTTP session support
- special HTTP transport settings
- transaction class mappings

Modify the following properties if you have a complex application:
1. If your Web application requires a virtual host, other than the default_host, or requires servlet caching, modify the Web container **General Properties**.
2. If your application requires persistent HTTP session support, modify the Web Container **Additional Properties > Session Management** setting.

3. If your application requires one of the following HTTP transport settings:
   - Unique hostname and port for client access
   - SSL enablement

   modify the Web Container **Additional Properties > HTTP transports** setting.
4. If your application requires global settings for internal servlets for WAR files packaged by third-party tools, modify the Web Container **Additional Properties > Custom Properties** setting.
5. If your application uses transaction class mappings to classify workload, modify the Web Container **Additional Properties > Advanced Settings**.

Web container

A Web container handles requests for servlets, JavaServer Pages (JSP) files, and other types of files that include server-side code. The Web container creates servlet instances, loads and unloads servlets, creates and manages request and response objects, and performs other servlet management tasks.

The Web server plug-ins, provided by the WebSphere Application Server, help supported Web servers pass servlet requests to Web containers.

Web container settings

Use this page to configure the web container settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server_instance* > **Web container**.

Configuration - General Properties

Default virtual host

Specifies a virtual host that enables a single host machine to resemble multiple host machines. Resources associated with one virtual host cannot share data with resources associated with another virtual host, even if the virtual hosts share the same physical machine.

Select a virtual host option:

Default Host
: The product provides a default virtual host with some common aliases, such as the machine IP address, short host name, and fully qualified host name. The alias comprises the first part of the path for accessing a resource such as a servlet. For example, it is `localhost:9080` in the request `http://localhost:9080/myServlet`.

Admin Host
: This is another name for the application server; also known as *server1* in the base installation. This process supports the use of the administrative console.

Servlet caching

Specifies that if a servlet is invoked once and it generates output to be cached, a cache entry is created containing not only the output, but also side effects of the invocation. These side effects can include calls to other servlets or Java Server Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information.

Enable servlet caching
: Check this box to enable servlet caching.

Web module settings

Use this page to configure Web module settings.

Access this page by traversing the following path in the administrative console:

Applications > **Enterprise Application** > *application_instance* > **Web Module**

URI

Specifies a URI that, when resolved relative to the application URL, specifies the location of the module archive contents on a file system. The URI must match the ModuleRef URI in the deployment descriptor of an application if the module was packaged as part of a deployed application or enterprise archive (EAR) file.

Name

Specifies the unique display name for the module.

Alternate DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module JAR file.

This file is the *post-assembly* version of the deployment descriptor file. You can edit the original deployment descriptor file to resolve dependencies and security information. Specifying the use of the alternative deployment descriptor keeps the original deployment descriptor file intact.

The value of the *Alternate DD* property must be the full path name of the deployment descriptor file, relative to the module root directory. By convention, the file is in the `ALT-INF` directory. If this property is not specified, the deployment descriptor file is read from the module JAR file.

Starting weight

Specifies the order in which modules are started. Lower weighted modules are started before higher weighted modules.

Prefer WEB-INF Classes

Specifies classes to load in WEB-INF before any other classes. Implementing the application class loader is recommended so that classes and resources packaged within the WAR file load before classes and resources residing in container-wide library JAR files.

Initial State

Specifies the default state of this application at server startup.

Web Module Deployment settings

Use this page to configure an instance of Web module deployment.

To view this administrative console page, click **Applications** > **Enterprise Application** > *application_instance* > **Web Modules** > *Web Module_instance*.

URI

Specifies a URI that, when resolved relative to the application URL, specifies the location of the module archive contents on a file system. The URI must match the ModuleRef URI in the deployment descriptor of an application if the module was packaged as part of a deployed application or enterprise archive (EAR) file.

Alternate DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module JAR file.

This file is the *post-assembly* version of the deployment descriptor file. You can edit the original deployment descriptor file to resolve dependencies and security information. Specifying the use of the alternative deployment descriptor keeps the original deployment descriptor file intact.

The value of the *Alternate DD* property must be the full path name of the deployment descriptor file, relative to the module root directory. By convention, the file is in the `ALT-INF` directory. If this property is not specified, the deployment descriptor file is read from the module JAR file.

Starting weight

Specifies the order in which modules are started. Lower weighted modules are started before higher weighted modules.

Classloader Mode

Specifies whether the class loader should search in the parent class loader or in the application class loader first to load a class. The standard for JDK class loaders and WebSphere class loaders is PARENT_FIRST. By specifying PARENT_LAST, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overriden classes and non-overriden classes.
The options are PARENT_FIRST and PARENT_LAST. The default is to search in the parent class loader before searching in the application class loader to load a class.

Data type	String
Default	PARENT_FIRST

Web container advanced settings

Use this page to support Web container advanced settings. This support includes Network QoS and transaction class mapping

To view this administrative console page, click **Servers** > **Application Servers** > *server name* > **Web Container** > **Advanced Settings**.

Network QoS

Specifies the parameter that will be used to classify outbound data that is delivered in response to HTTP and HTTPS requests.
The classification parameters are used to construct an ApplicationData parameter for the TCP/IP network service, which is called Quality of Service (QoS). The ApplicationData parameter is used in a QoS PolicyRule statement.

You can specify at most one classification parameter. If you do not specify a classification parameter, the response data will not be classified to the network agent.

Parameter	Description
HOST	Indicates that the Host value from the Host header, not including the port, is to be used used to construct an ApplicationData parameter. If you specify this parameter, WebSphere for z/OS classifies the outbound response data by using the HOST value. In the request: `http://www.mycompany.com/mywebap/myservlet` `www.mycompany.com` represents the host value.

URI
: Indicates that the part of the Universal Resource Locator that specifies the path to a resource is to be used to construct an ApplicationData parameter. If you specify this parameter, WebSphere for z/OS classifies the outbound response data by using the URI value. The path must be specified exactly as it is entered in a browser because the check for this path is case sensitive.

  In the request:

  ```
  http://www.mycompany.com/mywebap/myservlet
  ```

  `/mywebap/myservlet` represents the URI value.

HOSTURI
: Indicates that the HOST and URI, concatenated together, are to be used to construct an ApplicationData parameter. If you specify this parameter, WebSphere for z/OS classifies the outbound response using the concatenated HOST and URI value.

  In the request:

  ```
  Get request: http://www.mycompany.com/mywebap/myservlet
  ```

  `www.mycompany.com/mywebap/myservlet` represents the concatentated HOST and URI value.

TCLASS
: Indicates that a valid Workload Management (WLM) transaction class is to be used to construct an ApplicationData parameter. If you specify this parameter, you must specify the fully qualified name of the transaction class mapping file on the Transaction Class Mapping property.

Transaction Class Mapping

Specifies the fully qualified name of the file that contains the rules for classifying the Workload Management Transaction Class for HTTP or HTTPS requests. The file name is class sensitive.
For example, if `tclass.conf` is the name of your transaction class mapping file, you would specify the following for the value on this property:

```
/mydir/tclass.conf
```

where *mydir* is the fully qualified directory where the `tclass.conf` file is located.
For example
  /mydir/tclass.conf

Web container custom properties

Use this page to configure arbitrary name-value pairs of data, where the name is a property key and the value is a string value that can be used to set internal system configuration properties. Defining a new property enables you to configure a setting beyond that which is available in the administrative console.

To view this administrative console page, click **Servers > Application Servers >***server_name***> Web Container > Custom Properties**.

Name

Specifies the name (or key) for the property.

Data type	String

Value

Specifies the value paired with the specified name.

Data type	String

Description

Provides information about the name-value pair.

Data type	String

Global settings for internal servlets

Web Archive (WAR) files packaged using third-party tools cannot specify behavior for the services exposed by the Web container internal servlets. You can globally enable/disable internal servlets for all Web applications at the Web container level by creating name-value pairs such as:

Name	Value
fileServingEnabled	true
directoryBrowsingEnabled	true
serveServletsByClassnameEnabled	true

Settings defined at the Application Assembly Tool level take precedence over the global settings set through the custom properties at the Web container level.

Web application deployment extensions continue to hold configuration information for the services provided by the internal servlets, and take precedence over the global settings set through the custom properties at the Web container level.

UTF-8 encoded URLs

WebSphere Application Server Version 5.1, introduces support for UTF-8 encoded Uniform Resource Locators (URLs) to support the double byte characters in URLs. The UTF-8 encoded URL feature is enabled by default. You can prevent the web container from explicitly decoding URLs in UTF-8 and have them use the ISO-8859 standard as per the current HTTP specification by using the following name-value pair:

Name	Value
DecodeUrlAsUTF8	false

Web applications: Resources for learning

Use the following links to find relevant supplemental information about Web applications. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- "Web applications: Resources for learning"
- "Web applications: Resources for learning"
- "Web applications: Resources for learning"

Programming model and decisions

- J2EE BluePrints for Web applications
- Redbook on the design and implementation of Servlets, JSP files, and enterprise beans

Programming instructions and examples

- Redbook on Servlet and JSP file Programming
- Sun's Java™ Tutorial on Servlets
- Introduction to JavaServer Pages - Tutorial
- Bean Scripting Framework description
- Web delivered samples in the Samples Gallery

Programming specifications

- Java 2 Software Development Kit (SDK)
- Servlet 2.3 Specification
- JavaServer Pages 1.2 Specification
- Differences between JavaScript and ECMAScript
- ISO 8859 Specifications

Chapter 3. Managing HTTP sessions

IBM WebSphere Application Server provides a service for managing HTTP sessions: Session Manager. The key activities for session management are summarized below.

Before you begin these steps, make sure you are familiar with the programming model for accessing HTTP session support in the applications following the Servlet 2.3 API.

1. Plan your approach to session management, which could include session tracking, session recovery, and session clustering.
2. Create or modify your own applications to use session support to maintain sessions on behalf of Web applications.
3. 5.0.2 + Assemble your application.
4. 5.0.1 Assemble your application.
5. Deploy your application.
6. Ensure the administrator appropriately configures session management in the administrative domain.
7. Adjust configuration settings and perform other tuning activities for optimal use of sessions in your environment.

Sessions

A session is a series of requests to a servlet, originating from the same user at the same browser.

Sessions allow applications running in a Web container to keep track of individual users.

For example, a servlet might use sessions to provide "shopping carts" to online shoppers. Suppose the servlet is designed to record the items each shopper indicates he or she wants to purchase from the Web site. It is important that the servlet be able to associate incoming requests with particular shoppers. Otherwise, the servlet might mistakenly add Shopper_1's choices to the cart of Shopper_2.

A servlet distinguishes users by their unique session IDs. The session ID arrives with each request. If the user's browser is cookie-enabled, the session ID is stored as a cookie. As an alternative, the session ID can be conveyed to the servlet by URL rewriting, in which the session ID is appended to the URL of the servlet or JavaServer Pages (JSP) file from which the user is making requests. For requests over HTTPS or Secure Sockets Layer (SSL), Another alternative is to use SSL information to identify the session.

Migrating HTTP sessions

Note: In Version 5 default write frequency mode is TIME_BASED_WRITES, which is different from Version 4.0 and 3.5 default mode of END_OF_SERVICE.

Migrating from Version 4.0

No programmatic changes are required to migrate from version 4.0 to version 5.

Migrating from Version 3.5

If you have Version 3.5 applications running in Servlet 2.1 mode, some of the following Version 5 differences might influence how you choose to track and manage sessions.

1. During application development, modify session-related APIs as needed.

   Some API changes are required in order to redeploy existing applications on Version 5. These include changes to the HttpSession API itself as well as issues associated with moving to support for the Servlet 2.3 specification. Certain Servlet 2.1 API methods have been deprecated in Servlet 2.3 API . These deprecated APIs still work in Version 5.0, but they may be removed in a future version of the API. Changes are summarized in the following list:
   - Replace instances of getValue() with getAttribute()
   - Replace instances of getValueNames() with getAttributeNames()
   - Replace instances of removeValue() with removeAttribute()
   - Replace instances of putValue() with setAttribute()
2. During application development, modify Web application behavior as needed.

   In accordance with the Servlet 2.3 specification, HttpSession objects must be scoped within a single Web application context; they may not be shared between contexts. This means that a session can no longer span Web applications. Objects added to a session by a servlet or JSP in one Web application cannot be accessed from another Web application. The same session ID may be shared (because the same cookie is in use), but each Web application will have a unique session associated with the session ID. Version 5 provides a feature that can be used to extend scope of a session to enterprise application.
3. Use administrative tools to configure Session Manager security settings as needed. Relative to session security, the default Session Manager setting for Integrate Security is now false. This is different from the default setting in some earlier releases.
4. Use administrative tools to configure the JSP enabler and application server as needed.

   In Version 3.5 of the product, JSP files that contained the usebean tag with scope set to session did not always work properly when session persistence was enabled. Specifically, the JSP writer needed to write a scriplet to explicitly set the attribute (that is, to call setAttribute()) if it was changed as part of JSP processing.

   Two new features in Version 5.0 help address this problem:
   - You can set dosetattribute to true on the JSP InitParameter.
   - You can set the Write Contents option to Write all.

   The differences between the two solutions are summarized in the following table:

	Applies to	Configured at	Action
dosetattribute set to *true*	JSP	JSP enabler	Assures that JSP session-scoped beans always call setAttribute()
Write Contents option set to *Write all*	servlet or JSP	application server	All session data (changed or unchanged) is written to the external location

If session persistence is enabled and a class reload for the Web application occurs, the sessions associated with the Web application are maintained in the persistent store and will be available after the reload.

Developing session management in servlets

This information, combined with the coding example SessionSample.java, provides a programming model for implementing sessions in your own servlets.

1. Get the HttpSession object.

   To obtain a session, use the getSession() method of the javax.servlet.http.HttpServletRequest object in the Java Servlet 2.3 API.

   When you first obtain the HttpSession object, the Session Management facility uses one of three ways to establish tracking of the session: cookies, URL rewriting, or Secure Sockets Layer (SSL) information.

   Assume the Session Management facility uses cookies. In such a case, the Session Management facility creates a unique session ID and typically sends it back to the browser as a *cookie*. Each subsequent request from this user (at the same browser) passes the cookie containing the session ID, and the Session Management facility uses this ID to find the user's existing HttpSession object.

   In Step 1 of the code sample, the Boolean(create) is set to `true` so that the HttpSession object is created if it does not already exist. (With the Servlet 2.3 API, the javax.servlet.http.HttpServletRequest.getSession() method with no boolean defaults to `true` and creates a session if one does not already exist for this user.)

2. Store and retrieve user-defined data in the session.

   After a session is established, you can add and retrieve user-defined data to the session. The HttpSession object has methods similar to those in java.util.Dictionary for adding, retrieving, and removing arbitrary Java objects.

   In Step 2 of the code sample, the servlet reads an integer object from the HttpSession, increments it, and writes it back. You can use any name to identify values in the HttpSession object. The code sample uses the name sessiontest.counter.

   Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

3. (Optional) Output an HTML response page containing data from the HttpSession object.

4. Provide feedback to the user that an action has taken place during the session. You may want to pass HTML code to the client browser indicating that an action has occurred. For example, in step 3 of the code sample, the servlet

generates a Web page that is returned to the user and displays the value of the sessiontest.counter each time the user visits that Web page during the session.

5. (Optional) Notify Listeners. Objects stored in a session that implement the javax.servlet.http.HttpSessionBindingListener interface are notified when the session is preparing to end and become invalidated. This notice enables you to perform post-session processing, including permanently saving the data changes made during the session to a database.
6. End the session. You can end a session:
   - Automatically with the Session Management facility if a session is inactive for a specified time. The administrators provide a way to specify the amount of time after which to invalidate a session.
   - By coding the servlet to call the invalidate() method on the session object.

Example: SessionSample.java

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;


public class SessionSample  extends HttpServlet {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
       throws ServletException, IOException {


   // Step 1: Get the Session object

      boolean create = true;
      HttpSession session = request.getSession(create);

   // Step 2: Get the session data value


      Integer ival = (Integer)
      session.getAttribute ("sessiontest.counter");
      if (ival == null) ival = new Integer (1);
      else ival = new Integer (ival.intValue () + 1);
      session.setAttribute ("sessiontest.counter", ival);

   // Step 3: Output the page

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html>");
      out.println("<head><title>Session Tracking Test</title></head>");
      out.println("<body>");
      out.println("<h1>Session Tracking Test</h1>");
      out.println ("You have hit this page " + ival + " times" + "<br>");
      out.println ("Your " + request.getHeader("Cookie"));
      out.println("</body></html>");
   }
}
```

Assembling so that session data can be shared

In accordance with the Servlet 2.3 API specification, by default the Session Management facility supports session scoping by Web module. Only servlets in the same Web module can access the data associated with a particular session. WebSphere Application Server provides an option that you can use to extend the scope of the session attributes to an enterprise application. Therefore, you can

share session attributes across all the Web modules in an enterprise application. This option is provided as an IBM extension.

Restriction: To use this option, you must install all the Web modules in the enterprise application on a given server. You cannot split up Web modules in the enterprise application by servers. For example, with an enterprise application containing two Web modules, you cannot use this option when one Web module is installed on one server and second Web module is installed on a different server. In such split installations, applications might share session attributes across Web modules using distributed sessions, but session data integrity is lost when concurrent access to a session is made in different Web modules. It also severely restricts use of some Session Management features, like TIME_BASED_WRITES. For enterprise applications on which this option is enabled, the Session Management configuration on the Web module inside the enterprise application is ignored. Then Session Management configuration defined on enterprise application is used if Session Management is overwritten at the enterprise application level. Otherwise, the Session Management configuration on the Web container is used.

Servlet API Behavior

Note: If shared HttpSession context is turned on in an enterprise application, HttpSession listeners defined in all the Web modules inside the enterprise application are invoked for session events. The order of listener invocation is not guaranteed.

Do the following to share session data across Web modules in an enterprise application:

1. Start theAssembly ToolkitorApplication Assembly Tool (AAT).
2. In the Assembly Toolkit, right-click the application (EAR file) you want to share and click **Open With > Deployment Descriptor Editor**.In the AAT, click the EAR file that you want to share and click **IBM extension tab**.
3. In the application deployment descriptor editor of the Assembly Toolkit, select **Shared session context** under **WebSphere Extensions**.In the AAT, click **IBM extension tab > Shared httpsession context > Apply**. Make sure the class definition of attributes put into session are available to all Web modules in the enterprise application.
4. Save the application (EAR) file. In the Assembly Toolkit, after you close the application deployment descriptor editor, confirm that you want to save changes made to the application.

Session security support

You can integrate HTTP sessions and security in IBM WebSphere Application Server. When security integration is enabled in the Session Management facility and a session is accessed in a protected resource, you can access that session only in protected resources from then on. You cannot mix secured and unsecured resources accessing sessions when security integration is turned on. Security integration in the Session Management facility is not supported in form-based login with SWAM.

Security integration rules for HTTP sessions

Only authenticated users can access sessions created in secured pages and are created under the identity of the authenticated user. Only this authenticated user

can access these sessions in other secured pages. To protect these sessions from unauthorized users, you cannot access them from an unsecure page.

Programmatic details and scenarios

IBM WebSphere Application Server maintains the security of individual sessions.

An identity or user name, readable by the com.ibm.websphere.servlet.session.IBMSession interface, is associated with a session. An unauthenticated identity is denoted by the user name `anonymous`. IBM WebSphere Application Server includes the com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException class, which is used when a session is requested without the necessary credentials.

The Session Management facility uses the WebSphere Application Server security infrastructure to determine the authenticated identity associated with a client HTTP request that either retrieves or creates a session. WebSphere Application Server security determines identity using certificates, LPTA, and other methods.

After obtaining the identity of the current request, the Session Management facility determines whether to return the session requested using a getSession() call or not.

The following table lists possible scenarios in which security integration is enabled with outcomes dependent on whether the HTTP request is authenticated and whether a valid session ID and user name was passed to the Session Management facility.

	Unauthenticated HTTP request is used to retrieve a session	**HTTP request is authenticated, with an identity of ″FRED″ used to retrieve a session**
No session ID was passed in for this request, or the ID is for a session that is no longer valid	A new session is created. The user name is `anonymous`	A new session is created. The user name is `FRED`
A session ID for a valid session is passed in. The current session user name is ″anonymous″	The session is returned.	The session is returned. Session Management changes the user name to `FRED`
A session ID for a valid session is passed in. The current session user name is `FRED`	The session is not returned. An Unauthorized Session Request Exception error is thrown*	The session is returned.
A session ID for a valid session is passed in. The current session user name is `BOB`	The session is not returned. An Unauthorized Session Request Exception error is thrown*	The session is not returned. An Unauthorized Session Request Exception error is thrown*

* A com.ibm.websphere.servlet.session.UnauthorizedSessionRequestException error is thrown to the servlet.

Session management support

WebSphere Application Server provides facilities, grouped under the heading *Session Management*, that support the javax.servlet.http.HttpSession interface described in the Servlet API specification.

In accordance with the Servlet 2.3 API specification, the Session Management facility supports session scoping by Web module. Only servlets in the same Web module can access the data associated with a particular session. Multiple requests

from the same browser, each specifying a unique Web application, result in multiple sessions with a shared session ID. You can invalidate any of the sessions that share a session ID without affecting the other sessions.

You can configure a session timeout for each Web application. A Web application timeout value of 0 (the default value) means that the invalidation timeout value from the Session Management facility is used.

When an HTTP client interacts with a servlet, the state information associated with a series of client requests is represented as an HTTP session and identified by a session ID. Session Management is responsible for managing HTTP sessions, providing storage for session data, allocating session IDs, and tracking the session ID associated with each client request through the use of cookies or URL rewriting techniques. Session Management can store session-related information in several ways:

- In application server memory (the default). This information cannot be shared with other application servers.
- In a database. This storage option is known as *database persistent sessions*.
- In another WebSphere Application Server instance. This storage option is known as *memory-to-memory sessions*.

The last two options are referred to as *distributed sessions*. Distributed sessions are essential for using HTTP sessions for failover facility. When an application server receives a request associated with a session ID that it currently does not have in memory, it can obtain the required session state by accessing the external store (database or memory-to-memory). If distributed session support is not enabled, an application server cannot access session information for HTTP requests that are sent to servers other than the one where the session was originally created. Session Management implements caching optimizations to minimize the overhead of accessing the external store, especially when consecutive requests are routed to the same application server.

Storing session states in an external store also provides a degree of fault tolerance. If an application server goes offline, the state of its current sessions is still available in the external store. This availability enables other application servers to continue processing subsequent client requests associated with that session.

Saving session states to an external location does not completely guarantee their preservation in case of a server failure. For example, if a server fails while it is modifying the state of a session, some information is lost and subsequent processing using that session can be affected. However, this situation represents a very small period of time when there is a risk of losing session information.

The drawback to saving session states in an external store is that accessing the session state in an external location can use valuable system resources. Session Management can improve system performance by caching the session data at the server level. Multiple consecutive requests that are directed to the same server can find the required state data in the cache, reducing the number of times that the actual session state is accessed in external store and consequently reducing the overhead associated with external location access.

Configuring session management by level

When you configure session management at the Web container level, all applications and the respective Web modules in the Web container normally inherit that configuration, setting up a basic default configuration for the applications and Web modules below it.

However, you can set up different configurations individually for specific applications and Web modules that vary from the Web container default. These different configurations override the default for these applications and Web modules only.

Note: When you overwrite the default session management settings on the application level, all the Web modules below that application inherit this new setting unless they too are set to overwrite these settings.

1. Open the Administrative console.
2. Select the level that this configuration applies to:
   - For the web container level:
     a. Click **Servers** > **Application Servers**.
     b. Select a server from the list of application servers.
     c. Under Additional Properties, click **Web Container**.
   - For the Web module level:
     a. Click **Applications** > **Enterprise Applications**.
     b. Select an applications from the list of applications.
     c. Under Related Items, click **Web Modules**.
     d. Select a Web module from the list of Web modules defined for this application.
3. Under **Additional Properties**, click **Session Management**.
4. Make whatever changes you need to manage sessions
5. If you are working on the Web module or application level and want these settings to override the inherited Session Management settings, under **General Properties**, select **Overwrite**.
6. Click **Apply** and **Save**.

Session tracking options

There are several options for session tracking, depending on what sort of tracking method you want to use:
- Session tracking with cookies
- Session tracking with URL rewriting
- Session tracking with Secure Sockets Layer (SSL) information

Session tracking with cookies

Tracking sessions with cookies is the default. No special programming is required to track sessions with cookies.

Session tracking with URL rewriting

An application that uses URL rewriting to track sessions must adhere to certain programming guidelines. The application developer needs to do the following:
- Program servlets to encode URLs
- Supply a servlet or Java Server Pages (JSP) file as an entry point to the application

Using URL rewriting also requires that you enable URL rewriting in the Session Management facility.

Note: In certain cases, clients cannot accept cookies. Therefore, you cannot use cookies as a session tracking mechanism. Applications can use URL rewriting as a substitute.

Program session servlets to encode URLs

Depending on whether the servlet is returning URLs to the browser or redirecting them, include either encodeURL( ) or encodeRedirectURL( ) in the servlet code. Examples demonstrating what to replace in your current servlet code follow.

Rewrite URLs to return to the browser

Suppose you currently have this statement:

```
out.println("<a href=\"/store/catalog\">catalog<a>");
```

Change the servlet to call the encodeURL method before sending the URL to the output stream:

```
out.println("<a href=\"");
out.println(response.encodeURL ("/store/catalog"));
out.println("\">catalog</a>");
```

Rewrite URLs to redirect

Suppose you currently have the following statement:

```
response.sendRedirect ("http://myhost/store/catalog");
```

Change the servlet to call the encodeRedirectURL method before sending the URL to the output stream:

```
response.sendRedirect (response.encodeRedirectURL ("http://myhost/store/catalog"));
```

The encodeURL() and encodeRedirectURL() methods are part of the HttpServletResponse object. These calls check to see if URL rewriting is configured before encoding the URL. If it is not configured, the calls return the original URL.

If both cookies and URL rewriting are enabled and response.encodeURL() or encodeRedirectURL() is called, the URL is encoded, even if the browser making the HTTP request processed the session cookie.

You can also configure session support to enable protocol switch rewriting. When this option is enabled, the product encodes the URL with the session ID for switching between HTTP and HTTPS protocols.

Supply a servlet or JSP file as an entry point

The entry point to an application (such as the initial screen presented) may not require the use of sessions. However, if the application in general requires session support (meaning some part of it, such as a servlet, requires session support), then after a session is created, all URLs are encoded to perpetuate the session ID for the servlet (or other application component) requiring the session support.

The following example shows how you can embed Java code within a JSP file:

```
<%
response.encodeURL ("/store/catalog");
%>
```

Session tracking with SSL information

No special programming is required to track sessions with Secure Sockets Layer (SSL) information.

To use SSL information, turn on **Enable SSL ID tracking** in the Session Management property page. Because the SSL session ID is negotiated between the Web browser and HTTP server, this ID cannot survive an HTTP server failure. However, the failure of an application server does not affect the SSL session ID if an external HTTP Server is present between WebSphere Application Server and the browser.

SSL tracking is supported for the IBM HTTP Server and iPlanet Web servers only. You can control the lifetime of an SSL session ID by configuring options in the Web server. For example, in the IBM HTTP Server, set the configuration variable SSLV3TIMEOUT to provide an adequate lifetime for the SSL session ID. An interval that is too short can cause a premature termination of a session. Also, some Web browsers might have their own timers that affect the lifetime of the SSL session ID. These Web browsers may not leave the SSL session ID active long enough to serve as a useful mechanism for session tracking. Internal Http Server of WebSphere also supports SSL Tracking.

When using the SSL session ID as the session tracking mechanism in a cloned environment, use either cookies or URL rewriting to maintain session affinity. The cookie or rewritten URL contains session affinity information that enables the Web server to properly route a session back to the same server for each request.

Configuring session tracking

To configure session tracking, complete the following:

1. Go to the appropriate level of Session Management.
2. Specify which session tracking mechanism you want to pass the session ID between the browser and the servlet:
   - To track sessions with cookies, click **Enable Cookies**.

     To change the cookie settings, click **Modify**.
   - To track sessions with URL rewriting, click **Enable URL Rewriting**.

     If you want to enable protocol switch rewriting, click **Enable protocol switch rewriting.**
   - To track sessions with SSL information, click **Enable SSL ID tracking**.
3. Click **Apply**.
4. Click **Save**.
5. Define the session recovery characteristics.

Serializing access to session data

The Servlet API supports concurrent access to a session in a given server instance. WebSphere Application Server provides an option to prevent the concurrent access to a session in a given server instance so that concurrent modification of a session does not occur in a given server instance. This prevention is achieved by synchronizing the requests based on session. When this feature is turned on, a

session is obtained for the request before invoking the servlet and requests are synchronized by locking the session for the servlet execution time. Note that synchronization is based on the memory copy of session. So this feature cannot serialize requests across servers based on session when session affinity fails.

Restriction: Use this feature only when concurrent modification of the same session data is possible and is not desirable by the application. This feature has overhead of serializing the requests based on a session.

Do the following to synchronize session access:

1. Select the level of Session Management on which you want to serialize session access.
2. Under Serialize Session access, click **Allow serial access**.
3. In the Maximum wait time box, type the amount of time, in milliseconds, a servlet waits on a session before continuing execution. The default is 120000 milliseconds or two minutes.
4. Select **Allow access on timeout** if you want the servlet to gain access to the session and continue normal execution even if the session is still locked by another servlet. If you do not select this box, the servlet execution will abort when the session request times out.
5. Click **Apply**.
6. Click **Save**.

Session Management settings

Use this page to manage HTTP session support. This support includes specifying a session tracking mechanism, setting maximum in-memory session count, controlling overflow, and configuring session timeout.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management**.

Overwrite Session Management

Specifies whether or not these session management settings take precedence over those normally inherited from a higher level for the current application or web module.

By default, web modules inherit session management settings from the application level above it, and applications inherit session management settings from the web container level above it.

Session tracking mechanism

Specifies a mechanism for HTTP session management.

Mechanism	Function	Default

Enable SSL ID Tracking	Specifies that session tracking uses Secure Sockets Layer (SSL) information as a session ID. Enabling SSL tracking takes precedence over cookie-based session tracking and URL rewriting. There are two parameters available if you enable SSL ID tracking: SSLV3Timeout and Secure Authentication Service (SAS). SSLV3Timeout specifies the time interval after which SSL sessions are renegotiated. This is a high setting and modification does not provide any significant impact on performance. The SAS parameter establishes an SSL connection only if it goes out of the Java Virtual Machine (JVM) to another JVM. If all the beans are located within the same JVM, the SSL used by SAS does not hinder performance. These are set by editing the `sas.server.properties` and `sas.client.props` files located in the *install_root*\properties directory, where *install_root* is the directory where WebSphere Application Server is installed.	9600 seconds

Enable Cookies	Specifies that session tracking uses cookies to carry session IDs. If cookies are enabled, session tracking recognizes session IDs that arrive as cookies and tries to use cookies for sending session IDs. If cookies are not enabled, session tracking uses Uniform Resource Identifier (URL) rewriting instead of cookies (if URL rewriting is enabled). Enabling cookies takes precedence over URL rewriting. Do not disable cookies in the Session Management facility of the application server that is running the administrative application because this action causes the administrative application not to function after a restart of the server. As an alternative, run the administrative application in a separate process from your applications. Click **Modify** to change these settings.
Enable URL Rewriting	Specifies that the session management facility uses rewritten URLs to carry the session IDs. If URL rewriting is enabled, the session management facility recognizes session IDs that arrive in the URL if the encodeURL method is called in the servlet.
Enable Protocol Switch Rewriting	Specifies that the session ID is added to a URL when the URL requires a switch from HTTP to HTTPS or from HTTPS to HTTP. If rewriting is enabled, the session ID is required to go between HTTP and HTTPS.

Maximum in-memory session count

Specifies the maximum number of sessions to maintain in memory.

The meaning differs depending on whether you are using in-memory or distributed sessions. For in-memory sessions, this value specifies the number of sessions in the base session table. Use the Allow Overflow property to specify whether to limit sessions to this number for the entire Session Management facility or to allow additional sessions to be stored in secondary tables. For distributed

sessions, this value specifies the size of the memory cache for sessions. When the session cache has reached its maximum size and a new session is requested, the Session Management facility removes the least recently used session from the cache to make room for the new one.

Note: 5.0.2 + Do not set this value to a number less than the maximum thread pool size for your server.

Overflow

Specifies that the number of sessions in memory can exceed the value specified by the Max In Memory Session Count property. This option is valid only in nondistributed sessions mode.

Session timeout

Specifies how long a session can go unused before it is no longer valid. Specify either `Set timeout` or `No timeout`. Specify the value in minutes greater than or equal to two.

The value of this setting is used as a default when the session timeout is not specified in a Web module deployment descriptor. Note that to preserve performance, the invalidation timer is not accurate to the second. When the Write Frequency is time based, ensure that this value is least twice as large as the write interval.

Security integration

Specifies that when security integration is enabled, the session management facility associates the identity of users with their HTTP sessions

Serialize session access

Specifies that concurrent session access in a given server is not allowed.

Maximum wait time	Specifies the maximum amount of time a servlet request waits on an HTTP session before continuing execution. This parameter is optional and expressed in seconds. The default is 120, or 2 minutes. Under normal conditions, a servlet request waiting for access to an HTTP session gets notified by the request that currently owns the given HTTP session when the request finishes.
Allow access on timeout	Specifies whether the servlet is executed normally or aborted in the event of a timeout. If this box is checked, the servlet executes normally. If this box is not checked, the servlet execution aborts and error logs are generated.

Cookie settings

Use this page to configure cookie settings for session management.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Enable Cookies**.

Cookie name

Specifies a unique name for the session management cookie. The servlet specification requires the name JSESSIONID. However, for flexibility this value can be configured.

Secure cookies

Specifies that the session cookies include the secure field. Enabling the feature restricts the exchange of cookies to HTTPS sessions only.

Cookie domain

Specifies the domain field of a session tracking cookie. This value controls whether or not a browser sends a cookie to particular servers. For example, if you specify a particular domain, session cookies are sent to hosts in that domain. The default domain is the server.

Cookie path

Specifies that a cookie is sent to the URL designated in the path. Specify any string representing a path on the server. "/" indicates root directory. Specify a value to restrict the paths to which the cookie will be sent. By restricting paths, you prevent the cookie from going to certain URLs on the server. If you specify the root directory, the cookie is sent no matter which path on the given server is accessed.

Cookie maximum age

Specifies the amount of time that the cookie lives on the client browser. Specify that the cookie lives only as long as the current browser session, or to a maximum age. If you choose the maximum age option, specify the age in seconds. This value corresponds to the Time to Live (TTL) value described in the Cookie specification. Default is the current browser session which is equivalent to setting the value to -1.

Distributed sessions

WebSphere Application Server provides the following session mechanisms in a distributed environment:

- **Database Session persistence**, where sessions are stored in the database specified.
- **Memory-to-memory Session replication**, where sessions are stored in one or more specified WebSphere Application Server instances.

When a session contains attributes that implement HttpSessionActivationListener, notification occurs anytime the session is activated (that is, session is read to the memory cache) or passivated (that is, session leaves the memory cache). Passivation can occur because of a server shutdown or when the session memory cache is full and an older session is removed from the memory cache to make room for a newer session. It is not guaranteed that a session is passivated in one application server prior to being activated in another.

Session recovery support

For session recovery support, WebSphere Application Server provides distributed session support in the form of database sessions and memory-to-memory replication. Use session recovery support under the following conditions:

- When the user's session data must be maintained across a server restart
- When the user's session data is too valuable to lose through an unexpected server failure

All the attributes set in a session must implement java.io.Serializable if the session requires external storage. In general, consider making all objects held by a session serialized, even if immediate plans do not call for session recovery support. If the Web site grows, and session recovery support becomes necessary, the transition

occurs transparently to the application if the sessions only hold serialized objects. If not, a switch to session recovery support requires coding changes to make the session contents serialized.

Distributed Environment settings

Use this page to specify a type for saving a session in a distributed environment.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Distributed Environment Settings**.

Distributed Sessions

Specifies the type of distributed environment to be used for saving sessions.

None	Specifies that the session management facility discards the session data when the server shuts down.
Database	Specifies that the session management facility stores session information in the data source specified by the data source connection settings. Click **Database** to change these data source settings.
Memory to Memory Replication	Specifies that the session management facility stores the session information in a data source in memory. The session information is copied to other session management facilities for failure recovery. Click **Memory to Memory Replication** to specify the replicator to use and to change these memory to memory settings. *(For WebSphere Application Server Network Deployment only.)*

Configuring for database session persistence

To configure the session management facility for database session persistence, complete the following:

1. Define a JDBC provider.
2. Create a DB2 table in the z/OS DB2 database that will be used for session persistence.
3. Create a data source pointing to the z/OS DB2 database containing the DB2 table for session persistence, using the JDBC provider that you defined. The data source should be non-JTA, for example, non-XA enabled. Note the JNDI name of the data source. Under **Data Sources >** *datasource_name* **> Custom Properties**, make sure the correct database is entered for the value of the **databaseName** property. If necessary, contact your database administrator to verify the correct database name.
4. Go to the appropriate level of Session Management.
5. Click **Distributed Environment Settings**
6. Select and click **Database**.
7. Specify the Data Source JNDI name from step 3.
8. Switch to a multirow schema.
9. Click **OK**.
10. If you want to change the tuning parameters, click **Custom Tuning Parameters** and select a setting or customize one.

11. Click **Apply**.
12. Click **Save**.

Switching to a multirow schema

By default, a single session maps to a single row in the database table used to hold sessions. With this setup, there are hard limits to the amount of user-defined, application-specific data that WebSphere Application Server can access.

1. Modify the Session Management facility properties to switch from single to multirow schema.
2. Manually drop and recreate the database table or delete all the rows in the database table that the product uses to maintain HttpSession objects.

   See the *DB2 UDB for OS/390 and z/OS V7 Administration Guide* for a description of how to drop a DB2 database table.

   Creating a DB2 table for session persistence describes how to create a new DB2 database table.

Creating a DB2 table for session persistence

If you are using DB2 for session persistence, a DB2 table, in which session data will be collected, must be created and defined to the application server.

To create a DB2 table for collecting session data, do the following:

1. Have your DB2 Administrator create a DB2 database table for storing your session data. (For more information about creating DB2 databases see the *DB2 UDB for OS/390 and z/OS V7 Administration Guide*.)

   The table space in which the database table is created must be defined with row level locking (LOCKSIZE ROW). It should also have a page size that is large enough for the objects that will be stored in the table during a session. Following is an example of a table space definition with row level locking specified and a buffer pool page size of 32K:

   ```
   CREATE DATABASE database_name
     STOGROUP SYSDEFLT
     CCSID EBCDIC;

   CREATE TABLESPACE tablespace_name IN database_name
     USING STOGROUP group_name
     PRIQTY 512
     SECQTY 1024
     LOCKSIZE ROW
     BUFFERPOOL BP32K;
   ```

   The Session Manager will use the DB2 table defined within this table space to process the session data. This table must have the following format:

   ```
   CREATE TABLE database_name.table_name  (
     ID              VARCHAR(95) NOT NULL ,
     PROPID          VARCHAR(95) NOT NULL ,
     APPNAME         VARCHAR(64) ,
     LISTENERCNT     SMALLINT ,
     LASTACCESS      DECIMAL(19,0),
     CREATIONTIME    DECIMAL(19,0),
     MAXINACTIVETIME  INTEGER ,
     USERNAME        VARCHAR(256) ,
     SMALL           VARCHAR(3122)  FOR BIT DATA ,
     MEDIUM          VARCHAR(28869) FOR BIT DATA ,
   ```

```
LARGE           BLOB(2097152),
SESSROW         ROWID NOT NULL GENERATED ALWAYS
)
IN database_name.tablespace_name;
```

Note: The length attributes specified for VARCHAR in this example are not necessarily the values your DB2 Administrator should use for the DB2 table he is creating. See the DB2 SQL Reference for the version of DB2 you will be using for guidance in determining appropriate values for these length attributes for your installation.

A unique index must be created on the `ID` and `PROPID` columns of this table. The following is an example of the index definition:

```
CREATE UNIQUE INDEX database_name.index_name.
      database_name.table_name
      (ID     ASC,
      PROPID  ASC,
      APPNAME ASC);
```

Note:

a. At run time, the Session Manager will access the target table using the identity of the J2EE server in which the owning Web application is deployed. Any Web container that is configured to use persistent sessions should be granted both read and update access to the subject database table.
b. HTTP session processing uses the index defined using the `CREATE INDEX` statement to avoid database deadlocks. In some situations, such as when a relatively small table size is defined for the database, DB2 may decide not to use this index. When the index isn't used, database deadlocks can occur. If this situation occurs, see the DB2 Administration Guide for the version of DB2 you are using for recommendations on how to calculate the space required for an index, and adjust the size of the tables you are using accordingly.
c. It may be necessary to tune DB2 in order to make efficient use of the sessions database table and to avoid deadlocks when accessing it. Your DB2 Administrator should refer to the DB2 Administration Guide for specific information about tuning the version of DB2 you are using.

A large object (LOB) table space must be defined and an auxiliary table must be defined within that table space. The following is an example of the LOB table space definition:

```
CREATE LOB TABLESPACE LOB_tablespace_name IN database_name
    BUFFERPOOL BP32K
    USING STOGROUP group_name
    PRIQTY 512
    SECQTY 1024
    LOCKSIZE LOB;

CREATE AUX TABLE database_name.aux_table_name
    IN database_name.LOB_tablespace_name
    STORES database_name.table_name
    COLUMN LARGE;
```

An index must be created for this auxiliary table. The following is an example of the index definition:

```
CREATE INDEX database_name.aux_index_name ON
      database_name.aux_table_name;
```

2. Have your DB2 Administrator grant the the z/OS userID, under which the server region is running, the appropriate access to this DB2 table. For example,issue the following command to grant z/OS userID CBASRU1, under which the server region is running, access to the table SESSIONS contained in the database SESSDB:

```
GRANT ALL ON SESSDB.SESSIONS TO CBASRU1;
```

3. Use the administrative console to add the name of this DB2 table to the Web container's configuration properties:
   a. Open the administrative console.
   b. Click **Servers** > **Application Servers**.
   c. Select a server from the list of application servers.
   d. Under Additional Properties, click **Web Container**.
   e. Under Additional Properties, click **Custom Properties**.
   f. Check **SessionTableName** and then click **New**.
   g. In the **Value** field, enter the name of the DB2 Session Table if you are not using the default value **SESSION**. The name must be in the form *database_name.table_name*. For example, if the database name is SESSDB and the table name is SESSIONS, enter SESSDB.SESSIONS for **Value**.

      Optionally, you can update the description of this table in the **Description** field. For example, you might enter "Table name for HTTP session data."
   h. Click **Apply** > **Save**.

When the product is restarted, the Session Management facility creates the new SESSIONS table in the specified tablespace.

Database settings

Use this page to specify the settings for database session support.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Distributed Environment Settings** > **Database**.

Datasource JNDI Name

Specifies the datasource description
The JNDI name of the non-XA enabled data source from which Session Management obtains database connections. For example, if the JNDI name of the datasource is "jdbc/sessions", specify "jdbc/sessions." The data source represents a pool of database connections and a configuration for that pool (such as the pool size). The data source must already exist as a configured resource in the environment.

User ID

Specifies the user ID for database access

Password

Specifies the password for database access

Confirm Password

Specifies the password a second time to ensure it recorded correctly.

DB2 Row Size

Specifies the tablespace page size configured for the sessions table, if using a DB2 database. Possible values are 4, 8, 16, and 32 kilobytes (K). The default row size is 4K.

The default row size is 4K. In DB2, it can be updated to a larger value. This can help database performance in some environments. When this value is other than 4, you must specify Table Space Name to use. For 4K pages, the Table Space Name is optional.

Table Space Name

Specifies that tablespace to be used for the sessions table.
This value is required when the DB2 Page Size is other than 4K.

Multi row schema

Specifies that each instance of application data be placed in a separate row in the database, allowing larger amounts of data to be stored for each session. This action can yield better performance in certain usage scenarios. If using multirow schema is not enabled, instances of application data can be placed in the same row.

Multirow schema considerations

IBM WebSphere Application Server supports the use of a multirow schema option in which each piece of application specific data is stored in a separate row of the database. With this setup, the total amount of data you can place in a session is now bound only by the database capacities. The only practical limit that remains is the size of the session attribute object.

The multirow schema potentially has performance benefits in certain usage scenarios, such as when larger amounts of data are stored in the session but only small amounts are specifically accessed during a given servlet processing of an HTTP request. In such a scenario, avoiding unneeded Java object serialization is beneficial to performance.

Understand that switching between multirow and single row is not a trivial proposition.

In addition to allowing larger session records, using multirow schema can yield performance benefits. However, it requires a little work to switch from single-row to multirow schema, as shown in the instructions below.

Coding considerations and test environment

Consider configuring direct single-row usage to one database and multirow usage to another database while you verify which option suits your application needs. (Do this in code by switching the data source used; then monitor performance.)

Programming issue	Application scenario
Reasons to use single-row	• You can read or write all values with just one record read and write. • This takes up less space in a database because you are guaranteed that each session is only one record long.
Reasons **not** to use single-row	2-megabyte limit of stored data per session.

Programming issue	Application scenario
Reasons to use multirow	• The application can store an unlimited amount of data; that is, you are limited only by the size of the database and a 2-megabyte-per-record limit. • The application can read individual fields instead of the whole record. When large amounts of data are stored in the session but only small amounts are specifically accessed during servlet processing of an HTTP request, multirow sessions can improve performance by avoiding unneeded Java object serialization.
Reasons **not** to use multirow	If data is small in size, you probably do not want the extra overhead of multiple row reads when you can store everything in one row.

In the case of multirow usage, design your application data objects not to have references to each other, to prevent circular references. For example, suppose you are storing two objects A and B in the session using HttpSession.put(..) method, and A contains a reference to B. In the multirow case, because objects are stored in different rows of the database, when objects A and B are retrieved later, the object graph between A and B is different than stored. A and B behave as independent objects.

Memory-to-memory replication

WebSphere Application Server supports session replication to another WebSphere Application Server instance. This support is referred to as *memory-to-memory session replication*. In this mode, sessions can replicate to one or more WebSphere Application Server instances to address HTTP Session single point of failure (SPOF). This is a new alternative in IBM WebSphere Application Server, Version 5 to the existing saving of HTTP Session to a database.

The WebSphere Application Server instance in which the session is currently processed is referred to as the *owner of the session*. In a clustered environment, session affinity in the WebSphere Application Server plug-in routes the requests for a given session to the same server. If the current owner server instance of the session fails, then the WebSphere Application Server plug-in routes the requests to another appropriate server in the cluster. This server either retrieves the session from a server that has the backup copy of the session or it retrieves the session from its own backup copy table. The server now becomes the owner of the session and affinity is now maintained to this server.

When a session is created or updated in a WebSphere Application Server instance, the session is transferred (or replicated) through one of the replicator entries under the replication domain that is configured with the session management facility. This session potentially gets replicated to the WebSphere Application Server instances that are also connected to the same replicator domain. The mode and partitioning determine whether WebSphere Application Server instances in the same replication domain gets the session.

There are three possible modes. You can set up a WebSphere Application Server instance to run in:

- **Server mode:** Only store backup copies of other WebSphere Application Server sessions and not to send out copies of any session created in that particular server
- **Client mode:** Only broadcast or send out copies of the sessions it owns and not to receive backup copies of sessions from other servers
- **Both mode:** Simultaneously broadcast or send out copies of the sessions it owns and act as a backup table for sessions owned by other WebSphere Application Server instances

You can select the replication mode of server, client, or both when configuring the session management facility for memory-to-memory replication. The default is both. This storage option is controlled by the mode parameter.

With respect to mode, the following are the primary examples of memory-to-memory replication configuration:
- Peer-to-peer with a local replicator
- Peer-to-peer with remote replicators
- Client/server with remote replicators
- Client/server with isolated replicators

In a cluster, by default, sessions are replicated in all the servers in the cluster that are connected to the same replicator domain. This replication can be redundant if a large number of servers exist in a cluster. The session management facility has an option to partition the servers into groups when storing sessions.

Clustered session support

A clustered environment supports load balancing, where the workload is distributed among the application servers that compose the cluster. In a cluster environment, the same Web application must exist on each of the servers that can access the session. You can accomplish this setup by installing an application onto a cluster definition. Each of the servers in the group can then access the Web application

In a clustered environment, the Session Management facility requires an affinity mechanism so that all requests for a particular session are directed to the same application server instance in the cluster. This requirement conforms to the Servlet 2.3 specification in that multiple requests for a session cannot coexist in multiple application servers. One such solution provided by IBM WebSphere Application Server is *session affinity* in a cluster; this solution is available as part of the WebSphere Application Server plug-ins for Web servers. It also provides for better performance because the sessions are cached in memory. In clustered environments other than WebSphere Application Server clusters, you must use an affinity mechanism (for example, IBM WebSphere Edge Server affinity).

If one of the servers in the cluster fails, it is possible for the request to reroute to another server in the cluster. If distributed sessions support is enabled, the new server can access session data from the database or another WebSphere Application Server instance. You can retrieve the session data only if a new server has access to an external location from which it can retrieve the session.

Tuning session management

IBM WebSphere Application Server session support has features for tuning session performance and operating characteristics, particularly when sessions are configured in a distributed environment. These options support the administrator flexibility in determining the performance and failover characteristics for their environment.

The table summarizes the features, including whether they apply to sessions tracked in memory, in a database, with memory-to-memory replication, or all. Click a feature for details about the feature. Some features are easily manipulated using administrative settings; others require code or database changes.

Feature or option	Goal	Applies to sessions in memory, database, or memory-to-memory
Write frequency	Minimize database write operations.	Database and Memory-to-Memory
Session affinity	Access the session in the same application server instance.	All
Multirow schema	Fully utilize database capacities.	Database
Base in-memory session pool size	Fully utilize system capacity without overburdening system.	All
Write contents	Allow flexibility in determining what session data to write	Database and Memory-to-Memory
Scheduled invalidation	Minimize contention between session requests and invalidation of sessions by the Session Management facility. Minimize write operations to database for updates to last access time only.	Database and Memory-to-Memory
Tablespace and row size	Increase efficiency of write operations to database.	Database (DB2 only)

Configuring scheduled invalidation

Instead of relying on the periodic invalidation timer that runs on an interval based on the session timeout parameter, you can set specific times for the session management facility to scan for invalidated sessions in a distributed environment. When used with distributed sessions, this feature has the following benefits:

- You can schedule the scan for invalidated sessions for times of low application server activity, avoiding contention between invalidation scans of database or another WebSphere Application Server instance and read and write operations to service HTTP session requests.
- Significantly fewer external write operations can occur when running with the End of Service Method Write mode because the last access time of the session

does not need to be written out on each HTTP request. (Manual Update options and Time Based Write options already minimize the writing of the last access time.)

Usage considerations

- With scheduled invalidation configured, HttpSession timeouts are not strictly enforced. Instead, all invalidation processing is handled at the configured invalidation times.
- HttpSessionBindingListener processing is handled at the configured invalidation times unless the HttpSession.invalidate( ) method is explicitly called.
- The HttpSession.invalidate() method immediately invalidates the session from both the session cache and the external store.
- The periodic invalidation thread still runs with scheduled invalidation. If the current hour of the day does not match one of the configured hours, sessions that have exceeded the invalidation interval are removed from cache, but not from the external store. Another request for that session results in returning that session back into the cache.
- When the periodic invalidation thread runs during one of the configured hours, all sessions that have exceeded the invalidation interval are invalidated by removal from both the cache and the external store.
- The periodic invalidation thread can run more than once during an hour and does not necessarily run exactly at the top of the hour.
- If you specify the interval for the periodic invalidation thread using the HttpSessionReaperPollInterval custom property, do not specify a value of more than 3600 seconds (1 hour) to ensure that invalidation processing happens at least once during each hour.

Configuring write contents

In Session Management, you can configure which session data is written to the database or to another WebSphere instance, depending on whether you are using database pesistent sessions or memory to memory replication. This flexibility allows for fewer code changes for the JSP writer when the application will be operating in a clustered environment. The following options are available in Session Management for tuning what is to be written back:

- Write changed (the default) - Write only session data properties that have been updated through setAttribute() and removeAttribute() method calls.
- Write all - Write all session data properties.

The **Write all** setting might benefit servlet and JSP writers who change Java objects' states that reside as attributes in HttpSession and do not call HttpSession.setAttribute().

However, the use of **Write all** could result in more data being written back than is necessary. If this situation applies to you, consider combining the use of **Write all** with **Time-based write** to boost performance overall. As always, be sure to evaluate the advantages and disadvantages for your installation.

With either Write Contents setting, when a session is first created, complete session information is written, including all of the objects bound to the session. When using database session persistence, in subsequent session requests, what is written to the database depends on whether a single-row or multirow schema has been set for the session database, as follows:

Write Contents setting	Behavior with single-row schema	Behavior with multirow schema
Write changed	If any session attribute is updated, all objects bound to the session are written.	Only the session data modified through setAttribute() or removeAttribute() calls is written.
Write all	All bound session attributes are written.	All session attributes that currently reside in the cache are written. If the session has never left the cache, all session attributes are written.

1. Go to the appropriate level of Session Management.
2. Click Distributed Environment Settings
3. Click Custom Tuning Parameters.
4. Select Custom Settings, and click Modify.
5. Select the appropriate write contents setting.

Configuring write frequency

In the Session Management facility, you can configure the frequency for writing session data to the database or to a WebSphere instance, depending on whether you use database distributed sessions or memory-to-memory replication. This flexibility enables you to weigh session performance gains against varying degrees of failover support. The following options are available in the Session Management facility for tuning write frequency:

- **End of service servlet**- Write session data at the end of the servlet service() method call.
- **Manual update**- Write session data only when the servlet calls the IBMSession.sync() method.
- **Time based** (the default) - Write session data at periodic intervals, in seconds (called the *write interval*).

When a session is first created, session information is always written at the end of the service() call.

Using the time based write or manual update options can result in loss of data in failover scenarios since the backup copy of the session in the persistent store (for example, a database or another JVM) may not be in sync with the session in the session cache.

Base in-memory session pool size

The base in-memory session pool size number has different meanings, depending on session support configuration:

- With in-memory sessions, session access is optimized for up to this number of sessions.
- With distributed sessions (meaning, when sessions are stored in a database or in another WebSphere Application Server instance); it also specifies the cache size and the number of last access time updates saved in manual update mode.

For distributed sessions, when the session cache has reached its maximum size and a new session is requested, the Session Management facility removes the least recently used session from the cache to make room for the new one.

General memory requirements for the hardware system, and the usage characteristics of the e-business site, determines the optimum value.

Note that increasing the base in-memory session pool size can necessitate increasing the heap sizes of the Java processes for the corresponding WebSphere Application Servers.

Overflow in nondistributed sessions

By default, the number of sessions maintained in memory is specified by base in-memory session pool size. If you do not wish to place a limit on the number of sessions maintained in memory and allow overflow, set overflow to *true*.

Allowing an unlimited amount of sessions can potentially exhaust system memory and even allow for system sabotage. Someone could write a malicious program that continually hits your site and creates sessions, but ignores any cookies or encoded URLs and never utilizes the same session from one HTTP request to the next.

When overflow is disallowed, the Session Management facility still returns a session with the HttpServletRequest getSession(true) method when the memory limit is reached, and this is an invalid session that is not saved.

With the WebSphere Application Server extension to HttpSession, com.ibm.websphere.servlet.session.IBMSession, an isOverflow() method returns *true* if the session is such an invalid session. An application can check this status and react accordingly.

Controlling write operations

You can manually control when modified session data is written out to the database or to another WebSphere Application Server instance by using the sync() method in the com.ibm.websphere.servlet.session.IBMSession interface, which extends the javax.servlet.http.HttpSession interface. By calling the sync() method from the service() method of a servlet, you send any changes in the session to the external location. When *manual update*is selected as the write frequency mode, session data changes are written to an external location only if the application calls the sync() method. If the sync() method is not called, session data changes are lost when a session object leaves the server cache. When *end of service servlet* or *time based* is the write frequency mode, the session data changes are written out whenever the sync() method is called. If the sync() method is not called, changes are written out at the end of service method or on a time interval basis based on the write frequency mode selected.

```
IBMSession iSession = (IBMSession) request.getSession();
iSession.setAttribute("name", "Bob");

//force write to external store
iSession.sync( )
```

Tuning parameter settings

Use this page to set tuning parameters for distributed sessions.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Distributed Environment Settings** > **Custom Tuning Parameters**.

Tuning Level

Specifies that the session management facility provides certain predefined settings that affect performance.

Select one of these predefined settings or customize a setting. To customize a setting, select one of the predefined settings that comes closest to the setting desired, click **Custom settings**, make your changes, and then click **OK**.

Very high (optimize for performance)

Write frequency	Time based
Write interval	300 seconds
Write contents	Only updated attributes
Schedule sessions cleanup	true
First time of day default	0
Second time of day default	2

High

Write frequency	Time based
Write interval	300 seconds
Write Contents	All session attributes
Schedule sessions cleanup	false

Medium

Write frequency	End of servlet service
Write Contents	Only updated attributes
Schedule sessions cleanup	false

Low (optimize for failover)

Write frequency	End of servlet service
Write Contents	All session attributes
Schedule sessions cleanup	false

Custom settings

Write frequency default	Time based
Write interval default	10 seconds
Write contents default	All session attributes
Schedule sessions cleanup default	false

Tuning parameter custom settings

Use this page to customize tuning parameters for distributed sessions.

To view this administrative console page, click **Servers** > **Application Servers** > *server_name* > **Web Container** > **Session Management** > **Distributed Environment Settings** > **Custom Tuning Parameters** > **Custom settings**.

Write frequency

Specifies when the session is written to the persistent store.

End of servlet service
A session writes to a database or another WebSphere Application Server instance after the servlet completes execution.

Manual update
A programmatic sync on the IBMSession object is required to write the session data to the database or another WebSphere Application Server instance.

Time based
Session data writes to the database or another WebSphere Application Server instance based on the specified Write Interval value. Default: 10 seconds

Write contents

Specifies whether updated attributes are only written to the external location or all of the session attributes are written to the external location, regardless of whether or not they changed. The external location can be either a database or another application server instance.

Only updated attributes
Only updated attributes are written to the persistent store.

All session attribute
All attributes are written to the persistent store.

Schedule sessions cleanup

Specifies when to clean the invalid sessions from a database or another application server instance.

Specify distributed sessions cleanup schedule
Enables the scheduled invalidation process for cleaning up the invalidated HTTP sessions from the external location. Enable this option to reduce the number of updates to a database or another application server instance required to keep the HTTP sessions alive. When this option is not enabled, the invalidator process runs every few minutes to remove invalidated HTTP sessions.

When this option is enabled, specify the two hours of a day for the process to clean up the invalidated sessions in the external location. Specify the times when there is the least activity in the application servers. An external location can be either a database or another application server instance.

First Time of Day (0 - 23)
Indicates the first hour, in Greenwich Mean Time (GMT), during which the invalidated sessions are cleared from the external location. Specify this value as a positive integer between 0 and 23. This value is valid only when schedule invalidation is enabled.

Second Time of Day (0 - 23)
Indicates the second hour, in Greenwich Mean Time (GMT), during which the invalidated sessions are cleared from the external location. Specify this value as a positive integer between 0 and 23. This value is valid only when schedule invalidation is enabled.

Best practices for using HTTP Sessions

- **Enable Security integration for securing HTTP sessions**

  HTTP sessions are identified by session IDs. A session ID is a pseudo-random number generated at the runtime. Session hijacking is a known attack HTTP sessions and can be prevented if all the requests going over the network are enforced to be over a secure connection (meaning, HTTPS). But not every configuration in a customer environment enforces this constraint because of the performance impact of SSL connections. Due to this relaxed mode, HTTP session is vulnerable to hijacking and because of this vulnerability, WebSphere Application Server has the option to tightly integrate HTTP sessions and WebSphere Application Server security. Enable security in WebSphere Application Server so that the sessions are protected in a manner that only users who created the sessions are allowed to access them.
- **Release HttpSession objects using `javax.servlet.http.HttpSession.invalidate()` when finished.**

  HttpSession objects live inside the Web container until:
  - The application explicitly and programmatically releases it using the `javax.servlet.http.HttpSession.invalidate()` method; quite often, programmatic invalidation is part of an application logout function.
  - WebSphere Application Server destroys the allocated HttpSession when it expires (default = 1800 seconds or 30 minutes). The WebSphere Application Server can only maintain a certain number of HTTP sessions in memory based on Session Management settings. In case of distributed sessions, when maximum cache limit is reached in memory, the Session Management facility removes the least recently used (LRU) one from cache to make room for a session.

  .
- **Avoid trying to save and reuse the HttpSession object outside of each servlet or JSP file.**

  The HttpSession object is a function of the HttpRequest (you can get it only through the req.getSession() method), and a copy of it is valid only for the life of the service() method of the servlet or JSP file. You *cannot* cache the HttpSession object and refer to it outside the scope of a servlet or JSP file.
- **Implement the java.io.Serializable interface when developing new objects to be stored in the HTTP session.**

  This action allows the object to properly serialize when using distributed sessions. Without this extension, the object cannot serialize correctly and throws an error. An example of this follows:

  ```
  public class MyObject implements java.io.Serializable {...}
  ```

  Make sure all instance variable objects that are not marked transient are serializable.
- **The HTTPSession API does not dictate transactional behavior for sessions.**

  Distributed HTTPSession support does not guarantee transactional integrity of an attribute in a failover scenario or when session affinity is broken. Use transactionally aware resources like enterprise Java beans to guarantee the transaction integrity required by your application.
- **Ensure the Java objects you add to a session are in the correct class path.**

  If you add Java objects to a session, place the class files for those objects in the correct classpath (the Application Classpath if utilizing sharing across Web modules in an enterprise application, or the WebModule Classpath if using the Servlet 2.2-complaint session sharing) or in the directory containing other

servlets used in WebSphere Application Server. In the case of session clustering, this action applies to every node in the cluster.

Because the HttpSession object is shared among servlets that the user might access, consider adopting a site-wide naming convention to avoid conflicts.

- **Avoid storing large object graphs in the HttpSession object.**

  In most applications each servlet only requires a fraction of the total session data. However, by storing the data in the HttpSession object as one large object, an application forces WebSphere Application Server to process all of it each time.
- **Utilize Session Affinity to help achieve higher cache hits in the WebSphere Application Server.**

  WebSphere Application Server has functionality in the HTTP Server plug-in to help with session affinity. The plug-in will read the cookie data (or encoded URL) from the browser and helps direct the request to the appropriate application or clone based on the assigned session key. This functionality increases use of the in-memory cache and reduces hits to the database or another WebSphere Application Server instance
- **Maximize use of session affinity and avoid breaking affinity.**

  Using session affinity properly can enhance the performance of the WebSphere Application Server. Session affinity in the WebSphere Application Server environment is a way to maximize the in-memory cache of session objects and reduce the amount of reads to the database or another WebSphere Application Server instance. Session affinity works by caching the session objects in the server instance of the application with which a user is interacting. If the application is deployed in multiple servers of a server group, the application can direct the user to any one of the servers. If the users starts on server1 and then comes in on server2 a little later, the server must write all of the session information to the external location so that the server instance in which server2 is running can read the database. You can avoid this database read using session affinity. With session affinity, the user starts on server1 for the first request; then for every successive request, the user is directed back to server1. Server1 has to look only at the cache to get the session information; server1 never has to make a call to the session database to get the information.

  You can improve performance by not breaking session affinity. Some suggestions to help avoid breaking session affinity are:
  - Combine all Web applications into a single application server instance, if possible, and use modeling or cloning to provide failover support.
  - Create the session for the frame page, but do not create sessions for the pages within the frame when using multiframe JSP files. (See discussion later in this topic.)
- **When using multi-framed pages, follow these guidelines:**
  - Create a session in only one frame or before accessing any frame sets. For example, assuming there is no session already associated with the browser and a user accesses a multi-framed JSP file, the browser issues concurrent requests for the JSP files. Because the requests are not part of any session, the JSP files end up creating multiple sessions and all of the cookies are sent back to the browser. The browser honors only the last cookie that arrives. Therefore, only the client can retrieve the session associated with the last cookie. Creating a session before accessing multi-framed pages that utilize JSP files is recommended.
  - By default, JSPs get a HTTPSession using `request.getSession(true)` method. So by default JSPs create a new session if none exists for the client. Each JSP page in the browser is requesting a new session, but only one session is used per browser instance. A developer can use `<% @ page session="false" %>` to turn off the automatic session creation from the JSP files that will not access the session. Then if the page needs access to the session information, the

developer can use `<%HttpSession session = javax.servlet.http.HttpServletRequest.getSession(false); %>` to get the already existing session that was created by the original session creating JSP file. This action helps prevent breaking session affinity on the initial loading of the frame pages.
  - Update session data using only one frame. When using framesets, requests come into the HTTP server concurrently. Modifying session data within only one frame so that session changes are not overwritten by session changes in concurrent frameset is recommended.
  - Avoid using multi-framed JSP files where the frames point to different Web applications. This action results in losing the session created by another Web application because the JSESSIONID cookie from the first Web application gets overwritten by the JSESSIONID created by the second Web application.
- **Secure all of the pages (not just some) when applying security to servlets or JSP files that use sessions with security integration enabled, .**

  When it comes to security and sessions, it is all or nothing. It does not make sense to protect access to session state only part of the time. When security integration is enabled in the Session Management facility, all resources from which a session is created or accessed must be either secured or unsecured. You cannot mix secured and unsecured resources.

  The problem with securing only a couple of pages is that sessions created in secured pages are created under the identity of the authenticated user. Only the same user can access sessions in other secured pages. To protect these sessions from use by unauthorized users, you cannot access these sessions from an unsecure page. When a request from an unsecure page occurs, access is denied and an `UnauthorizedSessionRequestException` error is thrown. (`UnauthorizedSessionRequestException` is a runtime exception; it is logged for you.)
- **Use manual update and either the sync() method or time-based write in applications that read session data, and update infrequently.**

  With END_OF_SERVICE as write frequency, when an application uses sessions and anytime data is read from or written to that session, the LastAccess time field updates. If database sessions are used, a new write to the database is produced. This activity is a performance hit that you can avoid using the Manual Update option and having the record written back to the database only when data values update, not on every read or write of the record.

  To use manual update, turn it on in the Session Management Service. (See the tables above for location information.) Additionally, the application code must use the `com.ibm.websphere.servlet.session.IBMSession` class instead of the generic HttpSession. Within the IBMSession object there is a method called sync(). This method tells the WebSphere Application Server to write the data in the session object to the database. This activity helps the developer to improve overall performance by having the session information persist only when necessary.

  Note: An alternative to using the manual updates is to utilize the timed updates to persist data at different time intervals. This action provides similar results as the manual update scheme.
- Implement the following suggestions to achieve high performance:
  - If your applications do not change the session data frequently, use Manual Update and the sync() function (or timed interval update) to efficiently persist session information.

- Keep the amount of data stored in the session as small as possible. With the ease of using sessions to hold data, sometimes too much data is stored in the session objects. Determine a proper balance of data storage and performance to effectively use sessions.
  - If using database sessions, use a dedicated database for the session database. Avoid using the application database. This helps to avoid contention for JDBC connections and allows for better database performance.
  - If using memory to memory sessions, define replicators only on the servers and have the client attach to server replicator.
  - If using memory to memory sessions, employ partitioning (either group or single replica) as your clusters grow in size and scaling decreases.
  - Verify that you have the latest fix packs for the WebSphere Application Server.
- Utilize the following tools to help monitor session performance.
  - Run the com.ibm.servlet.personalization.sessiontracking.IBMTrackerDebug servlet. - To run this servlet, you must have the servlet invoker running in the Web application you want to run this from. Or, you can explicitly configure this servlet in the application you want to run.
  - Use the WebSphere Application Server Resource Analyzer which comes with WebSphere Application Server to monitor active sessions and statistics for the WebSphere Application Server environment.
  - Use database tracking tools such as "Monitoring" in DB2. (See the respective documentation for the database system used.)

Managing HTTP sessions: Resources for learning:

Use the following links to find relevant supplemental information about HTTP sessions. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:

Programming model and decisions
- Best practices
- HTTP Session Persistence Best Practices
- Improving session persistence performance with DB2
- Persistent client state HTTP cookies specification

Programming instructions and examples
- Java Servlet documentation, tutorials, and examples site

Programming specifications
- Java Servlet 2.3 API specification download site
- J2EE 1.3 specification download site

Chapter 4. Using enterprise beans in applications

1. Design a J2EE application and the enterprise beans that it needs. See ″Resources for learning″ for links to design information that is specific to enterprise beans.
2. Develop any enterprise beans that your application will use.
3. Prepare for assembly. For your EJB 2.x-compliant entity beans, decide on an appropriate access intent policy.
4. Assemble the beans using theAssembly ToolkitorApplication Assembly Tool (AAT) into one or more EJB modules. This includes setting security.
5. Assemble the modules into a J2EE application using the Assembly ToolkitorAAT .
6. 5.0.2 + For a given application server, update the EJB container configuration if needed for the application to be deployed, and determine if you want to batch commands batch commands or defer commands for container managed persistence.
7. Deploy the application in an application server.
8. Test the modules.
   - As needed, debug problems with the container.
   - Debug access and deployment problems.
9. Assemble the production application using theAssembly ToolkitorAAT.
10. Deploy the application to a production environment.
11. Manage the application:
    a. Manage installed EJB modules. After an application has been installed, you can manage its EJB modules individually through administrative console settings.
    b. Manage other aspects of the J2EE application.
12. Update the module and redeploy it using theAssembly ToolkitorAAT.
13. Tune the performance of the application. See Best practices for developing enterprise beans.

Enterprise beans

An enterprise bean is a Java component that can be combined with other resources to create J2EE applications. There are three types of enterprise beans, *entity* beans, *session* beans, and *message-driven* beans.

All beans reside in EJB containers, which provide an interface between the beans and the application server on which they reside.

Entity beans store permanent data. Entity beans with container-managed persistence (CMP) require connections to a form of persistent storage. This storage might be a database, an existing legacy application, a file, or other types of persistent storage. Entity beans with bean-managed persistence manage permanent data in whichever manner is defined in the bean code. This can include writing to databases or XML files, for example.

© Copyright IBM Corp. 2003

Session beans do not require database access, although they can obtain it indirectly as needed through entity beans. Session beans can also obtain direct access to databases (and other resources) through the use of resource references. Session beans can be either *stateful* or *stateless*.

New in the Enterprise JavaBeans (EJB) specification, version 2.0, message-driven beans enable asynchronous message servicing. The EJB container and a Java Message Service (JMS) provider work together to process messages. When a message arrives from another application component through JMS, the EJB container forwards it through an onMessage() call to a message-driven bean instance, which then processes the message. In other respects, message-driven beans are similar to stateless session beans.

Beans that require data access use *data sources*, which are administrative resources that define pools of connections to persistent storage mechanisms.

For more information about enterprise beans, see ″Resources for learning.″

Developing enterprise beans

Design a J2EE application and the enterprise beans that it needs.
- For general design information, see ″Resources for learning.″
- Before developing entity beans with container-managed persistence (CMP), read ″Concurrency control.″

There are two basic approaches to selecting tools for developing enterprise beans:
- You can use one of the available integrated development environments (IDEs). IDE tools automatically generate significant parts of the enterprise bean code and contain integrated tools for packaging and testing enterprise beans. The IBM WebSphere Application Developer product is the recommended IDE. For more information, see the documentation for that product.
- If you have decided to develop enterprise beans without an IDE, you need at least an ASCII text editor. You can also use a Java development tool that does not support enterprise bean development. You can then use tools available in the Java Software Development Kit (SDK) and in this product to assemble, test, and deploy the beans.

The following steps primarily support the second approach, development without an IDE.
1. If necessary, migrate any pre-existing code to the required version of the Enterprise JavaBeans (EJB) specification.
2. Write and compile the components of the enterprise bean.
   - At a minimum, an EJB 1.1 session bean requires a bean class, a home interface, and a remote interface. An EJB 1.1 entity bean requires a bean class, a primary-key class, a home interface, and a remote interface.
   - At a minimum, an EJB 2.0 session bean requires a bean class, a home or local home interface, and a remote or local interface. An EJB 2.0 entity bean requires a bean class, a primary-key class, a remote home or local home interface, and a remote or local interface. The types of interfaces go together: If you implement a local interface, you must define a local home interface as well.

   Note: Optionally, the primary-key class can be *unknown*. See unknown primary-key class for more information.

- Available only through EJB 2.0, a message-driven bean requires only a bean class.
3. For each entity bean, complete work to handle persistence operations.
   - Create a database schema for the entity bean's persistent data.
     - For entity beans with container-managed persistence(CMP), you must store the bean's persistent data in one of the supported databases. The Application Assembly Tool automatically generates SQL code for creating database tables for CMP entity beans. If your CMP beans require complex database mappings, it is recommended that you use the IBM WebSphere Studio Application Developer product to generate code for the database tables.
     - For entity beans with bean-managed persistence (BMP), you can create the database and database table by using the database tools or use an existing database and database table.

     For more information on creating databases and database tables, consult your database documentation.
   - **(CMP entity beans for EJB 2.0 only)** Define finder queries with EJB Query Language (EJB QL).

     With EJB QL, you define finders in terms of CMP fields and container-managed relationships, as follows:
     - *Public* finders are visible in the bean's home interface. Implemented in the bean class, they return only remote interfaces and collection types.
     - *Private* finders, expressed as SELECT statements, are used only within the bean class. They can return both local and remote interfaces, dependent values, other CMP field types, and collection types.
   - **(CMP entity beans for EJB 1.1 only: an IBM extension)** Create a finder helper interface for each CMP entity bean that contains specialized finder methods (other than the findByPrimaryKey method).

     The following logic is required for each finder method (other than the findByPrimaryKey method) contained in the home interface of an entity bean with CMP:
     - The logic must be defined in a public interface named `*Name*BeanFinderHelper`, where *Name* is the name of the enterprise bean (for example, AccountBeanFinderHelper).
     - The logic must be contained in a String constant named `*findMethodName* WhereClause`, where *findMethodName* is the name of the finder method. The String constant can contain zero or more question marks (?) that are replaced from left to right with the value of the finder method's arguments when that method is called.

5.0.1 **5.0.2** Assemble the beans in one or more EJB modules.

Migrating enterprise bean code to the supported specification

Support for Version 2.0 of the Enterprise JavaBeans (EJB) specification is new for Version 5 of this product. Migration of enterprise beans deployed in Version 4.0.x of this product is not generally necessary; Version 1.1 of the EJB specification is still supported. Follow these steps as appropriate for your application deployment.

1. Modify enterprise bean code for changes in the specification.
   - For Version 1.0 beans, migrate at least to Version 1.1.

- As stated previously, migration from Version 1.1 to Version 2.0 of the EJB specification is not required for redeployment on this version of the product. However, if your application requires the capabilities of Version 2.0, migrate your Version 1.1-compliant code.

  Note: The EJB Version 2.0 specification mandates that prior to the EJB container's executing a findBy*Method* query, the state of all enterprise beans enlisted in the current transaction be synchronized with the persistent store. (This is so the query is performed against current data.) If Version 1.1 beans are reassembled into an EJB 2.0-compliant module, the EJB container synchronizes the state of Version 1.1 beans as well as that of Version 2.0 beans. As a result, you might notice some change in application behavior even though the application code for the Version 1.1 beans has not been changed.

2. Reassemble and redeploy all modules to incorporate migrated code.

Migrating enterprise bean code from Version 1.0 to Version 1.1

The following information generally applies to any enterprise bean that currently complies with Version 1.0 of the Enterprise JavaBeans (EJB) specification. For more information about migrating code for beans produced with the IBM WebSphere Studio Application Developer tool, see the documentation for that product. For more information about migrating code in general, see ″Resources for learning.″

1. In session beans, replace all uses of javax.jts.UserTransaction with javax.transaction.UserTransaction. Entity beans may no longer use the UserTransaction interface at all.
2. In finder methods for entity beans, include FinderException in the `throws` clause.
3. Remove throws of java.rmi.RemoteException; throw javax.ejb.EJBException instead. However, continue to include RemoteException in the `throws` clause of home and remote interfaces as required by the use of Remote Method Invocation (RMI).
4. Remove uses of the finalize() method.
5. Replace calls to getCallerIdentity() with calls to getCallerPrincipal(). The use of getCallerIdentity() is deprecated.
6. Replace calls to isCallerInRole(Identity) with calls to isCallerinRole (String). The use of isCallerInRole(Identity) and java.security.Identity is deprecated.
7. Replace calls to getEnvironment() in favor of JNDI lookup. As an example, change the following code:

   ```
   String homeName =
      aLink.getEntityContext().getEnvironment().getProperty("TARGET_HOME_NAME");
   if (homeName == null) homeName = "TARGET_HOME_NAME";
   ```

   The updated code would look something like the following:

   ```
   Context env = (Context)(new InitialContext()).lookup("java:comp/env");
   String homeName = (String)env.lookup("ejb10-properties/TARGET_HOME_NAME");
   ```

8. In ejbCreate methods for an entity bean with container-managed persistence (CMP), return the bean's primary key class instead of `void`.
9. Add the getHomeHandle() method to home interfaces.

   ```
   public javax.ejb.HomeHandle getHomeHandle() {return null;}
   ```

Consider enhancements to match the following changes in the specification:
- Primary keys for entity beans can be of type java.lang.String.
- Finder methods for entity beans return java.util.Collection.

- Check the format of any JNDI names being used. Local name spaces are also supported.
- Security is defined by role, and isolation levels are defined at the method level rather than at the bean level.

Migrating enterprise bean code from Version 1.1 to Version 2.0

Enterprise JavaBeans (EJB) Version 2.0-compliant beans may be assembled only in an EJB 2.0-compliant module, although an EJB 2.0-compliant module can contain a mixture of Version 1.x and Version 2.0 beans.

The EJB Version 2.0 specification mandates that prior to the EJB container's executing a findBy*Method* query, the state of all enterprise beans enlisted in the current transaction be synchronized with the persistent store. (This is so the query is performed against current data.) If Version 1.1 beans are reassembled into an EJB 2.0-compliant module, the EJB container synchronizes the state of Version 1.1 beans as well as that of Version 2.0 beans. As a result, you might notice some change in application behavior even though the application code for the Version 1.1 beans has not been changed.

The following information generally applies to any enterprise bean that currently complies with Version 1.1 of the EJB specification. For more information about migrating code for beans produced with the IBM WebSphere Studio Application Developer tool, see the documentation for that product. For more information about migrating code in general, see "Resources for learning."

1. In beans with container-managed persistence (CMP) version 1.x, replace each CMP field with abstract get and set methods. In doing so, you must make each bean class abstract.
2. In beans with CMP version 1.x, change all occurrences of `this.`*`field`*` = `*`value`* to `set`*`Field`*`(`*`value`*`)`.
3. In each CMP bean, create abstract get and set methods for the primary key.
4. In beans with CMP version 1.x, create an EJB Query Language statement for each finder method.
5. In finder methods for beans with CMP version 1.x, return java.util.Collection instead of java.util.Enumeration.
6. Update handling of non-application exceptions.
   - To report non-application exceptions, throw javax.ejb.EJBException instead of java.rmi.RemoteException.
   - Modify rollback behavior as needed: In EJB versions 1.1 and 2.0, all non-application exceptions thrown by the bean instance result in the rollback of the transaction in which the instance is running; the instance is discarded. In EJB 1.0, the container does not roll back the transaction or discard the instance if it throws java.rmi.RemoteException.
7. Update rollback behavior as the result of application exceptions.
   - In EJB versions 1.1 and 2.0, an application exception does not cause the EJB container to automatically roll back a transaction.
   - In EJB Version 1.1, the container performs the rollback only if the instance has called setRollbackOnly() on its EJBContext object.
   - In EJB Version 1.0, the container is required to roll back a transaction when an application exception is passed through a transaction boundary started by the container.

WebSphere extensions to the Enterprise JavaBeans specification

This article outlines extensions to the Enterprise JavaBeans (EJB) specification that IBM provides with this product:

Inheritance in enterprise beans

In the Java language, *inheritance* is the creation of a new class from an existing class or a new interface from an existing interface. This product supports two forms of inheritance: standard class inheritance and EJB inheritance.

In standard class inheritance, the home interface, remote interface, or enterprise bean class inherits properties and methods from base classes that are not themselves enterprise bean classes or interfaces.

By contrast in enterprise bean inheritance, an enterprise bean inherits properties (such as container-managed persistence (CMP) fields and container-managed relationship (CMR) fields), methods, and method-level control descriptor attributes from another enterprise bean.

For more information, see the documentation for the IBM WebSphere Studio Application Developer product.

Optimistic concurrency control for container-managed persistence

This product supports optimistic concurrency control of data access.

Access intents for EJB persistence

5.0.2 + This product supports the application of named data-access policies.

Performance enhancements

Through the lifetime-in-cache settings, this product provides a way for you to improve performance for beans that are only occasionally updated. For more information, see "Entity bean assembly settings."

Some enterprise beans created with the IBM WebSphere Studio Application Developer product can utilize *read-ahead* for loading a bean and its related beans in a single database operation. An entire object graph or any part of the graph can be preloaded by configuring a finder method to use read-ahead.

Assembly and deployment extensions

5.0.1 5.0.2 This product supports IBM extensions of assembly and deployment options. IBM extensions are clearly marked in reference topics for assembly settings.

Best practices for developing enterprise beans

Use the following guidelines when designing and developing enterprise beans:

- Use a stateless session bean to act as the entry point for business logic. For more information about using session facades, see "Resources for learning."
- Entity beans should use container-managed persistence.

- In an Enterprise JavaBeans (EJB) Version 2.0 environment, use local interfaces to improve communication between enterprise beans in the same Java virtual machine.

  Local calls avoid the overhead of RMI/IIOP and use pass-by-reference semantics instead of pass-by-value. For each call, the caller and callee beans share the state of arguments. EJB 2.0 beans can have both a local and remote interface but more typically have one or the other.
- For communicating with remote clients, provide remote and remote home interfaces. For communicating with local clients like servlets, entity beans, and message-driven beans, provide local and local home interfaces.

Batch commands for container managed persistence

From JDBC 2.0 on, *PreparedStatement* objects can maintain a list of commands that can be submitted together as a batch. Instead of multiple database round trips, there can be only one database round trip for all the batched persistence requests.

The WebSphere Application Server version 5.0.2 enables you to take advantage of this. You can turn this option on from the EJB CMP side. When you choose this option, the run time defers *ejbStore/ejbCreate/ejbRemove* or the equivalent database persistence requests (insert/update/delete) until they are needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. When the persistence operation finally happens, run time accumulates the database requests and uses JDBC *PreparedStatement* batch operation to make a single JDBC call for multiple rows of the same operation.

Setting the run time for batched commands:

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *Dcom.ibm.ws.pm.batch=true*.

Deferred Create for container managed persistence

The specification for Enterprise Java Beans (EJB) 2.x states that for Container Managed Persistence (CMP) during the *ejbCreate*, the container can create the representation of the entity in the database immediately, or defer it to a later time.

The WebSphere Application Server version 5.0.2 enables you to take advantage of this specification. You can turn this option on from the EJB CMP side. When you choose this option, the runtime defers *ejbCreate* (or the equivalent database persistence request) until it is needed. This can be at the end of the transaction, or when a flush is needed for finders related to this EJB type. By doing this you can reduce two round trips for the newly created entity (insert and update) to one (insert).

Setting the run time for deferred create:

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.

6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *Dcom.ibm.ws.pm.deferredcreate=true*.

Explicit invalidation in the Persistence Manager cache

Container managed persistence (CMP) entity beans can be configured as *long-lifetime* beans. A long-lifetime bean is one that is configured with *Lifetime In Cache Usage* equal to a value other than the default **OFF** (refer to Entity bean assembly settings). A value other than **OFF** means that data for this bean is cached beyond the end of the transaction in which the bean was obtained by a finder or other method. The *Lifetime In Cache Usage* and *Lifetime In Cache* values control the basic length of time the cached data remains valid. When the specified time runs out, the cached data is no longer valid. See the *LifetimeInCache* help sections of the Assembly Toolkit (ATK) for more details.

However, there is also an API that lets the client application code explicitly invalidate the cached data of a bean on demand, superceding the basic lifetime of the cache data as controlled by the *Lifetime In Cache Usage* and *Lifetime In Cache* settings. This is useful where an application that does not use CMP beans modifies the data that underlies a CMP bean (for example, it updates a database table to which a CMP bean is mapped). Such an application can inform WebSphere Application Server that any cached version of this bean data is **stale** and no longer matches what is in the database. The data should be invalidated (in essence, discarded). For CMP beans that cannot tolerate stale data, this is an important feature.

Because the PM Cache Invalidation mechanism does consume resources in exchange for its benefits, it is not enabled by default. To enable it refer to Setting Persistence Manager Cache Invalidation .

Example: Explicit Invalidation in the Persistence Manager Cache: Usage Scenario

The scenario of use for this feature begins with configuring one or more bean types to be long-lifetime beans (see Explicit Invalidation in the Persistence Manager Cache, and configuring the necessary Java Message Service (JMS) resources (described below). Once this is done, the server is started. The scenario continues as follows:

1. Assume that a CMP entity bean of type *Department* has been configured to be a long-lifetime bean.
2. Transaction 1 begins. Application code looks up *Department*'s home and calls a finder method (such as *findByPrimaryKey("dept01")*). As this is the first finder to return *Department dept01*, a trip is made to the database to obtain the data. Transaction 1 ends.
3. Transaction 2 begins. Application code calls *findByPrimaryKey("dept01")* again. Because this is not the first finder to return *Department dept01*, we get a cache hit and no database trip is made. So far this is current WebSphere Application Server behavior for long-lifetime beans. Transaction 2 ends.
4. Another application, which does not use the *Department* CMP bean, is executed. This application might or might not be run on the WebSphere Application Server; it could be a legacy application. The application updates the database table that is mapped to the *Department* bean, altering the row for *dept01*. For example, the *budget* column in the table (mapped to a Java double CMP attribute in the *Deparment* bean) is changed from $10,000.00 to $50,000.00. This application was designed to cooperate with WebSphere Application Server. After performing the update, the application sends an invalidate request message to invalidate the *Department* bean *dept01*.

5. Transaction 3 begins. Application code looks up *Department*'s home and calls a finder method (such as *findByPrimaryKey("dept01")*). Because this is the first finder after *Department dept01* is invalidated, a new database trip is made to obtain the data. Transaction 3 ends.

Persistence Manager cache invalidation API

The PM cache invalidation API is in the form of a JMS message that the client sends to a specially-named JMS topic using a connection from a specifically named JMS *TopicConnectionFactory*. The JMS message must be an *ObjectMessage* created by the client. The client code creates a *PMCacheInvalidationRequest* object that describes the bean data to invalidate. Client code places the *PMCacheInvalidationRequest* object in the *ObjectMessage* and publishes the *ObjectMessage* (for further details on the JMS objects and terms used here, please see the Java Message Service documentation).

The public class *PMCacheInvalidationRequest* is central to the API, so we include a portion of its code here for illustration purposes (if you see any differences between this illustration and the actual class, the class is to be considered correct):

```
packagecom.ibm.websphere.ejbpersistence;

/**
*An instance of this class represents a request to invalidate one or more
*CMP beans in the PMcache.When an invalidate occurs,cached datafor this
*bean is removed from the cache;the next time an application tries to find
*this bean,a fresh copy of the bean data is obtained from the data store.
*
*The ability to invalidate a bean means that a CMP bean may be configured
*as a long-lifetime bean and thus be cached across transactions for much
*greater performance on future attempts to find this bean.Yet when some
*outside mechanism updates the bean data,sending an invalidation request
*will remove stale data from the PMcache so applications do not behave falsely
*based on stale data.
*/
public class PMCacheInvalidationRequestimplementsSerializable{

. . .

/**
 * Constructor used to invalidate a single bean
 * @param beanHomeJNDIName the JNDI name of the bean home. This is the same value
 * used to look up the bean home prior to calling findByPrimaryKey, for example.
 * @param beanKey the primary key of the bean to be invalidated. The actual
 * object type must be the primary key type for this bean type.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName, Object beanKey)
 throws IOException {
. . .
}
/**
 * Constructor used to invalidate a Collection of beans
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 * @param beanKeys a Collection of the primary keys of the beans to be
 * invalidated. The actual type of each object in the Collection must be the
 * primary key type for this bean type.
*/
 public PMCacheInvalidationRequest(String beanHomeJNDIName, Collection beanKeys)
 throws IOException {
. . .
}
/**
```

```
 * Constructor used to invalidate all beans of a given type
 * @param beanHomeJNDIName java.lang.String the JNDI name of the bean home.
 * This is the same value used to look up the bean home prior to calling
 * findByPrimaryKey, for example.
 */
public PMCacheInvalidationRequest(String beanHomeJNDIName) {
 . . .
}

}
```

If the client wants to perform the invalidation in a synchronous way, it can opt to receive an acknowledgement JMS message when the invalidation is complete. To ask for an acknowledgement (ACK) message, the client sets a *Topic* of its own choosing in the *JMSReplyTo* field of the *ObjectMessage* for the invalidation request (see JMS documentation for further details). The client then waits (using the *receive()* method of JMS) on receipt of the acknowledgement message before continuing execution.

An ACK message enables the caller to insure there is not even a brief (seconds or less) window during which PM cache data is stale. The sending of an acknowledgement for each request does, of course, take a bit more time and so is recommended to be used only when needed.

The JMS resources used to make an invalidation request (*TopicConnectionFactory*, *TopicDestination*, and so forth) must be configured by the user (using the Administration console or other method) if they want to use PM Cache Invalidation. In this way the user can chose whichever JMS provider they prefer (as long as it supports pub-sub). The names that must be used for these resources are defined as part of the API, and use names unique to the WebSphere Application Server namespace to avoid name conflict with customer JMS resources.

The following are the names that must be used when the user configures the JMS resources (shown as Java constants for clarity):

```
// The JNDI name of the TopicConnectionFactory
 private static final String topicConnectionFactoryJNDIName =
  "com.ibm.websphere.ejbpersistence.InvalidateTCF";
// The JNDI name of the TopicDestination
  private static final String topicDestinationJNDIName =
  "com.ibm.websphere.ejbpersistence.invalidate";
// The Topic name (part of the TopicDestination)
  private static final String topicString =
  "com.ibm.websphere.ejbpersistence.invalidate";
```

Here are examples of how these constants can be used in client code:

```
// Look up the TopicConnectionFactory...
InitialContext ic = new InitialContext();
TopicConnectionFactory topicConnectionFactory =
 (TopicConnectionFactory) ic.lookup(topicConnectionFactoryJNDIName);
...
// Look up the Topic
Topic topic = (Topic) ic.lookup(topicDestinationJNDIName);
```

Note that JMS messages can be sent not only from J2EE application code (for example, a SessionBean or BMP entity bean method) but also from non-J2EE applications if your chosen JMS provider allows for this. For example, the IBM MQ provider, available in WebSphere Application Server as the **Embedded Messaging** feature (selectable during installation), supports the use of MQ classes (or structures in other languages) to create a topic connection and topic that are

compatible with the *TopicConnectionFactory* and *TopicDestination* you configure using WebSphere Application Server Application Console.

Setting Persistence Manager Cache Invalidation:

1. Open the administrative console.
2. Select **Servers**.
3. Select **Application Servers**.
4. Select the server you want to configure.
5. In the Additional Properties area, select **Process Definition**.
6. In the Additional Properties area, select **Java Virtual Machine**.
7. Update the **Generic JVM arguments** with *-Dcom.ibm.ws.ejbpersistence.cacheinvalidation=true*.

Unknown primary-key class

When writing an entity bean for Enterprise Java Bean Version 2.0, the minimum requirements usually include a primary-key class. However, in some cases you might choose not to specify the primary-key class for an entity bean with container managed persistence (CMP). Perhaps there is no obvious primary key, or you want to allow the deployer to select the primary key fields at deployment time. The primary key type is usually derived from the type used by the database system that stores the entity objects, and you might not know what this key is.

So, the *unknown key type* is actually a type chosen at deployment time, making it changeable each time the bean is deployed. Your client code must deal with this key as type *Object*.

Currently, WebSphere Application Server supports top-down mapping and enables the deployer to choose *String* keys generated at the application server. For an example of how to use this function, see the Samples library.

Using access intent policies

5.0.2 + You can use access intent policies to help the product run-time environment manage various aspects of Enterprise JavaBeans (EJB) persistence. You apply access intent policies to EJB Version 2.0 entity beans and their methods by using an application assembly tool. A set of default access intent policies comes with the Assembly Toolkit and Application Assembly Tool (AAT).

1. **5.0.2 +** Apply default access intent to CMP entity beans. For more information, see the online help available with the Assembly Toolkit or the Entity bean assembly settings.
2. Apply access intent policies to methods of CMP entity beans.

Access intent policies

An access intent policy is a named set of properties (access intents) that governs data access for Enterprise JavaBeans (EJB) persistence. You can assign policies to an entity bean and to individual methods on an entity bean's home, remote, or local interfaces during assembly. Access intents are settable only within EJB Version 2.x-compliant modules for entity beans with CMP Version 2.x.

This product supplies a number of access intent policies that specify permutations of read intent and concurrency control; the pessimistic/update policy can be

qualified further. The selected policy determines the appropriate isolation level and locking strategy used by the run-time environment.

Access intent policies are specifically designed to supplant the use of isolation level and access intent method-level modifiers found in the extended deployment descriptor for EJB version 1.1 enterprise beans. You cannot specify isolation level and read-only modifiers for EJB version 2.0 enterprise beans.

5.0.2 + Access intent policies configured on an entity basis define the default access intent for that entity. The default access intent is used to control that entity in the absence of a more specific configuration based upon either method-level policy configuration or application profiling.

5.0.2 + Access intent can be controlled in a more precise way by using either application profiling or by using method-level access intent policies. Application profiling is only available in the WebSphere Application Server Enterprise product. Method-level access intent policies are named and defined at the module level. A module can have one or many such policies. Policies are assigned, and apply, to individual methods of the declared interfaces of entity beans and their associated home interfaces. A method-based policy is acted upon by the combination of the EJB container and persistence manager when the method causes the entity to load.

For entity beans that are backed by tables with nullable columns, use an optimistic policy with caution. Nullable columns are automatically excluded from overqualified updates at deployment time; concurrent changes to a nullable field might result in lost updates. When used with the IBM WebSphere Studio Application Developer product, this product provides support for selecting a subset of the nonnullable columns that are to be reflected in the overqualified update statement that is generated in the deployment code to support optimistic policies.

5.0.2 + An entity that is configured with a read-only policy that causes a bean to be activated can cause problems if updates are attempted within the same transaction. Those changes will not be committed, and an exception will be thrown because data integrity might be compromised.

Concurrency control

Concurrency control is the management of contention for data resources. A concurrency control scheme is considered *pessimistic* when it locks a given resource early in the data-access transaction and does not release it until the transaction is closed. A concurrency control scheme is considered *optimistic* when locks are acquired and released over a very short period of time at the end of a transaction. The objective of optimistic concurrency is to minimize the time over which a given resource would be unavailable for use by other transactions. This is especially important with long-running transactions, which under a pessimistic scheme would lock up a resource for unacceptably long periods of time.

Under an optimistic scheme, locks are obtained immediately before a read operation and released immediately afterwards. Update locks are obtained immediately before an update operation and held until the end of the transaction.

To enable optimistic concurrency, this product uses an *overqualified update scheme* to test whether the underlying data source has been updated by another transaction since the beginning of the current transaction. With this scheme, the columns marked for update and their original values are added explicitly through a

WHERE clause in the UPDATE statement so that the statement fails if the underlying column values have been changed. As a result, this scheme can provide column-level concurrency control; pessimistic schemes can control concurrency at the row level only.

Optimistic schemes typically perform this type of test only at the end of a transaction. If the underlying columns have not been updated since the beginning of the transaction, pending updates to container-managed persistence fields are committed and the locks are released. If locks cannot be acquired or if some other transaction has updated the columns since the beginning of the current transaction, the transaction is rolled back: All work performed within the transaction is lost.

Pessimistic and optimistic concurrency schemes require different transaction isolation levels. Enterprise beans that participate in the same transaction and require different concurrency control schemes cannot operate on the same underlying data connection.

Whether or not to use optimistic concurrency depends on the type of transaction. Transactions with a high penalty for failure might be better managed with a pessimistic scheme. (A high-penalty transaction is one for which recovery would be risky or resource-intensive.) For low-penalty transactions, it is often worth the risk of failure to gain efficiency through the use of an optimistic scheme. In general, optimistic concurrency is more efficient when update collisions are expected to be infrequent; pessimistic concurrency is more efficient when update collisions are expected to occur often.

Read-ahead hints

Read-ahead schemes enable applications to minimize the number of database roundtrips by retrieving a working set of container-managed persistence (CMP) beans for the transaction within one query. Read-ahead involves activating the requested CMP beans and caching the data for their related beans, which ensures that data is present for the beans that are most likely to be needed next by an application. A *read-ahead hint* is a canonical representation of the related beans that are to be read. It is associated with the *findByPrimaryKey* method for the requested bean type, which must be an EJB 2.x-compliant CMP entity bean.

5.0.2 + Read-ahead hints can be set only using the WebSphere Application Server Enterprise assembly tool or through the Add Access Intent wizard of the IBM WebSphere Studio Application Developer product.

5.0.2 + Read-ahead is only supported for access intent policies that can be applied by the backend against which the application is deployed. Otherwise, the read-ahead hint is disregarded.

5.0.2 + Currently, only findByPrimaryKey methods can have read-ahead hints. Only beans related to the requested beans by a container-managed relationship (CMR), either directly or indirectly through other beans, can be read ahead. Beans that use EJB inheritance should not be used in a read-ahead hint.

A read-ahead hint takes the form of a character string. You do not have to provide the string; the wizard generates it for you based on CMRs defined for the bean. The following example is provided as supplemental information only.

Suppose a CMP bean type A has a finder method that returns instances of bean A. A read-ahead hint for this method is specified using the following notation:
`RelB.RelC; RelD`

Interpret the preceding notation as follows:
- Bean type A has a CMR with bean types B and D.
- Bean type B has a CMR with bean type C.

For each bean of type A that is retrieved from the database, its directly-related B and D beans and its indirectly-related C beans are also retrieved. The order of the retrieved bean data columns in each row of the result set is the same as their order in the read-ahead hint: an A bean, a B bean (or null), a C bean (or null), a D bean (or null). For hints in which the same relationship is mentioned more than once (for example, *RelB.RelC*;*RelB.RelE*), a bean's data columns appear only once, at the position it first appears in the hint.

The tokens shown in the notation (*RelB* and so on) must be CMR field names for the relationships as defined in the deployment descriptor for the bean. In indirect relationships such as *RelB.RelC*, *RelC* is a CMR field name defined in the deployment descriptor for bean type B.

A single read-ahead hint cannot refer to the same bean type in more than one relationship. For example, if a Department bean has a relationship *employees* with the Employee bean and also has a relationship *manager* with the Employee bean, the read-ahead hint cannot specify both *employees* and *manager*.

For more information about how to set read-ahead hints, see the documentation for the Websphere Studio Application Developer product.

Applying access intent policies to methods

You apply an access intent policy to a method, or set of methods, in an application's entity beans through theAssembly ToolkitorApplication Assembly Tool (AAT).

1. Start theAssembly ToolkitorApplication Assembly Tool (AAT) .
2. Create or edit the application EAR file. For example, to change attributes of an existing application, select **File > Open**, then select the EAR file.
3. Select **EJB Modules >** ***moduleName*** **> Access Intent**.
4. To configure a new access intent policy, right-click and select **New**.
5. On the **New Access Intent** panel, specify a name and a description. These attributes are provided as a convenience to the developer and are not used at run time.
6. To select the methods to which the access intent policy should apply, click **Add** beside the Methods table.
7. From the **Applied access intent** list, select an access intent policy.
8. To override an attribute defined in the applied policy, click **Add** beside the Access intent attribute overrides table.
9. Click **OK** to exit the New Access Intent panel.
10. Save your configuration by selecting **File > Save**.

Access intent exceptions

The following exceptions are thrown in response to the application of access intent policies:

com.ibm.ws.ejbpersistence.utilpm.PersistenceManagerException
: If the method that drives the ejbLoad() method is configured to be read-only but updates are then made within the transaction that loaded the bean's state, an exception is thrown during invocation of the ejbStore()

method, and the transaction is rolled back. Likewise, the ejbRemove() method cannot succeed in a transaction that is set as read-only. If an update hint is applied to methods of entity beans with bean-managed persistence, the same behavior and exception results. The forwarded exception object contains the message string `PMGR1103E: update instance level read only bean` *beanName*

This exception is also thrown if the applied access intent policy cannot be honored because a finder, ejbSelect, or container-managed relationship (CMR) accessor method returns an inherently read-only result. The forwarded exception object contains the message string `PMGR1001: No such DataAccessSpec -` *methodName*

The most common occurrence of this error is when a custom finder that contains a read-only EJB Query Language (EJB QL) statement is called with an applied access intent of `wsPessimisticUpdate` or `wsPessimisticUpdate-Exclusive`. These policies require the use of a FOR UPDATE clause on the SQL SELECT statement to be executed, but a read-only query cannot support FOR UPDATE. Other examples of read-only queries include joins; the use of ORDER BY, GROUP BY, and DISTINCT keywords.

To eliminate the exception, edit the EJB query so that it does not return an inherently read-only result or change the access intent policy being applied.

- If an update access is required, change the applied access intent setting to `wsPessimisticUpdate-WeakestLockAtLoad` or `wsOptimisticUpdate`.
- If update access is not truly required, use `wsPessimisticRead` or `wsOptimisticRead`.
- If connection sharing between entity beans is required, use `wsPessimisticUpdate-WeakestLockAtLoad` or `wsPessimisticRead`.

com.ibm.websphere.ejb.container.CollectionCannotBeFurtherAccessed
If a lazy collection is driven after it is no longer in scope, and beyond what has already been locally buffered, a CollectionCannotBeFurtherAccessed exception is thrown.

com.ibm.ws.exception.RuntimeWarning
If an application is configured incorrectly, a run-time warning exception is thrown as the application starts; startup is ended. You can validate an application's configuration by choosing the verify function in the WebSphere Application Assembly Tool. Some examples of misconfiguration include:

- A method configured with two different access intent policies
- A method configured with an undefined access intent policy

javax.ejb.NoSuchEntityException
If an update fails under optimistic concurrency because fields changed within another transaction between load and store requests, a NoSuchEntityException is raised and the commit fails.

Access intent assembly settings

Access intent policies contain data-access settings for use by the persistence manager. Default access intent policies are configured on the entity bean. Optionally, you can associate access intent policies with one or more methods.

These settings are applicable only for EJB 2.x-compliant entity beans that are packaged in EJB 2.x-compliant modules. Connection sharing between beans with bean-managed persistence and those with container-managed persistence is possible if they all use the same access intent policy.

Name

Specifies a name for a mapping between an access intent policy and one or more methods.

Description

Contains text that describes the mapping.

Methods - Name

Specifies the name of an enterprise bean method, or the asterisk character (*). The asterisk is used to denote all of the methods of an enterprise bean's remote and home interfaces.

Methods - Enterprise bean

Specifies which enterprise bean contains the methods indicated in the Name setting.

Methods - Type

Used to distinguish between a method with the same signature that is defined in both the home and remote interface. Use `Unspecified` if an access intent policy applies to all methods of the bean.

Data type	String
Range	Valid values are `Home`, `Remote`,`Local`, `LocalHome` or `Unspecified`

Methods - Parameters

Contains a list of fully qualified Java type names of the method parameters. This setting is used to identify a single method among multiple methods with an overloaded method name.

Applied access intent

Specifies how the container must manage data access for persistence. Configurable both as a default access intent for an entity and as part of a method-level access intent policy.

Data type	String
Default	`wsPessimisticUpdate-WeakestLockAtLoad`. With Oracle, this is the same as wsPessimisticUpdate.
Range	Valid settings are `wsPessimisticUpdate`, `wsPessimisticUpdate-NoCollision`, `wsPessimisticUpdate-Exclusive`, `wsPessimisticUpdate-WeakestLockAtLoad`, `wsPessimisticRead`, `wsOptimisticUpdate`, or `wsOptimisticRead`. Only `wsPessimisticRead` and `wsOptimisticRead` are valid when class-level caching is enabled in the EJB container.

This product supports lazy collections. For each segment of a collection, iterating through the collection (*next()*) does not trigger a remote method call to retrieve the next remote reference. Two policies (`wsPessimisticUpdate` and `wsPessimisticUpdate-Exclusive`) are extremely lazy; the collection increment size is set to 1 to avoid overlocking the application. The other policies have a collection increment size of 25.

5.0.2 + If an entity is not configured with an access intent policy, the run-time environment typically uses `wsPessimisticUpdate-WeakestLockAtLoad` by default. If, however, the **Lifetime in cache** property is set on the bean, the default value of **Applied access intent** is `wsOptimisticRead`; updates are not permitted.

Additional information about valid settings follows:

Profile name	Concurrency control	Access type	Transaction isolation
wsPessimisticRead (Note 1)	pessimistic	read	For Oracle, read committed. Otherwise, repeatable read
wsPessimisticUpdate (Note 2)	pessimistic	update	For Oracle, read committed. Otherwise, repeatable read
wsPessimisticUpdate-Exclusive (Note 3)	pessimistic	update	serializable
wsPessimisticUpdate-NoCollision (Note 4)	pessimistic	update	read committed
wsPessimisticUpdate-WeakestLockAtLoad (Note 5)	pessimistic	update	Repeatable read
wsOptimisticRead	optimistic	read	read committed
wsOptimisticUpdate (Note 6)	optimistic	update	read committed

Notes:
1. Read locks are held for the duration of the transaction.
2. The generated SELECT FOR UPDATE query grabs locks at the beginning of the transaction.
3. SELECT FOR UPDATE is generated; locks are held for the duration of the transaction.
4. 5.0.2 + A plain SELECT query is generated. No locks are held, but updates are permitted. Use cautiously. This intent enables execution without concurrency control.
5. 5.0.2 + Where supported by the backend, the generated SELECT query does not include `FOR UPDATE`; locks are escalated by the persistent store at storage time if updates were made. Otherwise, the same as wsPessimisticUpdate.
6. Generated overqualified-update query forces failure if CMP column values have changed since the beginning of the transaction.

   5.0.2 + Be sure to review the rules for forming overqualified-update query predicates. Certain column types (for example, BLOB) are ineligible for inclusion in the overqualified-update query predicate and might affect your design.

Access intent best practices

This topic outlines issues to consider when applying access intent policies to Enterprise JavaBeans (EJB) methods.

- 5.0.2 + **Start by configuring the default access intent policy for an entity.** After your application is built and running, you can more finely tune certain access paths in your application using application profiling or method-level access intent.
- 5.0.2 + **Don't mix access types.** Avoid using both pessimistic and optimistic policies in the same transaction. For most databases, pessimistic and optimistic

policies use different isolation levels. This can result in multiple database connections, which prevents you from taking advantage of the performance benefits possible through connection sharing.

- **Take care when applying `wsPessimisticUpdate-NoCollision`.** This policy does not ensure data integrity. No database locks are held, so concurrent transactions can overwrite each other's updates. Use this policy only if you can be sure that only one transaction will attempt to update persistent store at any given time.

Frequently asked questions: Access intent

I have not applied any access intent policies at all. My application runs just fine with a DB2 database, but it fails with an Oracle database with the following message: `com.ibm.ws.ejbpersistence.utilpm.PersistenceManagerException: PMGR1001E: No such DataAccessSpec :FindAllCustomers. The backend datastore does not support the SQLStatement needed by this AccessIntent: (pessimistic update-weakestLockAtLoad)(collections: transaction/25) (resource manager prefetch: 0) (AccessIntentImpl@d23690a)`. Why?

If you have not configured access intent, all of your data is accessed under the default access intent policy (`wsPessimisticUpdate-WeakestLockAtLoad`). On DB2 databases, the weakest lock is a shared one, and the query runs without a FOR UPDATE clause. On Oracle databases, however, the weakest lock is an update lock; this means that the SQL query must contain a FOR UPDATE clause. However, not every SQL statement necessarily supports FOR UPDATE; for example, if the query is being run against multiple tables in a join, FOR UPDATE is not supported.

To avoid this problem, try either of the following:

- Modify your SQL query or reconfigure your application so that an update lock is supported
- Apply an access intent policy that supports optimistic concurrency

I am calling a finder method and I get an InconsistentAccessIntentException at run time. Why?

This can occur when you use method-level access intent policies to apply more control over how a bean instance is loaded. This execption indicates that the entity bean was previously loaded in the same transaction. This could happen if you called a multifinder method that returned the bean instance with access intent policy X applied; you are now trying to load the second bean again by calling its findByPrimaryKey method with access intent Y applied. Both methods must have the same access intent policy applied.

Likewise, if the entity was loaded once in the transaction using an access intent policy configured on a finder, you might have called a container-managed relationship (CMR) accessor method that returned the entity bean configured to load using that entity's default access intent.

To avoid this problem, ensure that your code does not load the same bean instance twice within the same transaction with different access intent policies applied. Avoid the use of method-level access intent unless absolutely necessary.

I have two beans in a container-managed relationship. I call findByPrimaryKey() on the first bean and then call getBean2( ), a CMR accessor method, on the returned instance. At that point, I get an InconsistentAccessIntentException. Why?

You are probably using read-ahead. When you loaded the first bean, you caused the second bean to be loaded under the access intent policy applied to the finder method for the first bean. However, you have configured your CMR accessor method from the first bean to the second with a different access intent policy. CMR accessor methods are really finder

methods in disguise; the run-time environment behaves as if you were trying to change the access intent for an instance you have already read from persistent store.

To avoid this problem, beans configured in a read-ahead hint are all driven to load with the same access intent policy as the bean to which the read-ahead hint is applied.

I have a bean with a one-to-many relationship to a second bean. The first bean has a pessimistic-update intent policy applied. When I try to add an instance of the second bean to the first bean's collection, I get an UpdateCannotProceedWithIntegrityException. Why?

The second bean probably has a read intent policy applied. When you add the second bean to the first bean's collection, you are not updating the first bean's state, you are implicitly modifying the second bean's state. (The second bean contains a foreign key to the first bean, which is modified.)

To avoid this problem, ensure that both ends of the relationship have an update intent policy applied if you expect to change the relationship at run time.

EJB modules

An EJB module is used to assemble one or more enterprise beans into a single deployable unit. An EJB module is stored in a standard Java archive (JAR) file.

An EJB module contains the following:
- One or more deployable enterprise beans.
- A deployment descriptor, stored in an Extensible Markup Language (XML) file. This file declares the contents of the module, defines the structure and external dependencies of the beans in the module, and describes how the beans are to be used at run time.

An EJB module can be used as a stand-alone application, or it can be combined with other EJB modules, or with Web modules, to create a J2EE application. An EJB module is installed and run in an enterprise bean container.

For more information about EJB modules, see "Resources for learning."

Assembling EJB modules

Assemble an Enterprise JavaBeans (EJB) module to contain enterprise beans and related code artifacts. Group Web components, client code, and resource adapter code in separate modules. After assembling an EJB module, you can install it as a stand-alone application or combine it with other modules into an enterprise application.

To increase performance, break container-managed persistence (CMP) enterprise beans into several enterprise bean modules during assembly. The load time for hundreds of beans is improved by distributing the beans across several JAR files and packaging them to an EAR file. Load time is faster when the administrative server attempts to start the beans, for example, 8-10 minutes versus more than one hour when one JAR file is used.

Use the Assembly Toolkit to assemble an EJB module in any of the following ways:
- Import an existing EJB module (EJB JAR file).
- Create a new EJB module.

- Copy code artifacts (such as entity beans) from one EJB module into a new EJB module.

1. Start the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
4. Migrate enterprise bean (JAR) files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import your enterprise bean files to the Assembly Toolkit.
5. Create a new EJB module.
6. Copy code artifacts (such as entity beans) from one EJB module into a new EJB module.
7. Verify the contents of the new EJB module in either of the following ways:
   - In the J2EE Hierarchy view, expand **EJB Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the EJB module in a Navigator view.

Assembling EJB modules

If you want to use existing Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 modules in your J2EE Version 1.3 application, migrate them to the Version 1.3 specification first.

Assemble an Enterprise JavaBeans (EJB) module to contain enterprise beans and related code artifacts. Group Web components, client code, and resource adapter code in separate modules.

An EJB module is installed as a stand-alone application or is combined with other modules into an enterprise application.

To increase performance, break CMP enterprise beans into several enterprise bean modules during assembly. The load time for hundreds of beans is improved by distributing the beans across several JAR files and packaging them to an EAR file. Load time is faster when the administrative server attempts to start the beans, for example, 8-10 minutes versus more than one hour when one JAR file is used.

The Application Assembly Tool (AAT) provides flexibility in assembling EJB modules. Options described below include:

- Importing an existing EJB module (EJB JAR file)
- Creating a new EJB module
- Copying code artifacts (such as entity beans) from one EJB module into a new EJB module

1. Start the AAT.
2. From the New tab, select **EJB Module**. Click **OK**. The navigation tree displays various sets of properties for configuring the new EJB module.
3. Use the property dialog shown in the AAT workspace to change the default file name and location.
   a. It is recommended that you change the display name so that it differs from the file name.

b. If you like, change the temporary location of the EJB module from the default location, *install_root*/bin.
4. Add at least one EJB component to the module.
   - Add at least one enterprise bean to the EJB component.
     - Import an existing JAR or EAR file containing EJB components.
       a. In the Navigation pane, right-click the **EJB Components** icon.
       b. Select **Import** from the pop-up menu.
       c. Click **Browse** to locate the archive file to import.
       d. Click **Open** to display the contents of the archive file. The applications in the selected archive file display.
       e. Select an EJB application from the archive file.
       f. Select the servlets or JSP files to be added, and click **Add** to display the components in the Selected Components window.
       g. Click **OK** to add the selected components.
     - Copy and paste values from an existing module.
     - Create a new EJB component.
       a. In the Navigation pane, right-click the **EJB Components** icon.
       b. Select **New** from the pop-up menu.
       c. Enter the component name and archive type.
       d. Select the class files.
       e. Click **OK** in the **New EJB Component** property dialog.
       f. Enter properties for the EJB component as needed.
5. Enter assembly properties for each bean.
   a. Click the plus sign (+) next to the component instance to show property groups.
   b. Right-click the icon for a property group.
   c. Select **New** from the pop-up menu to add new values, or edit existing values in the property pane.

   If you change the session type (`stateful` or `stateless`) of a session bean, you must click **Apply** before making any other changes to the same bean. Otherwise, certain input fields on the GUI become inactive. (You will know they are inactive because they are grayed out on the GUI.) After making all of your changes, click **Apply** again to commit them.
6. Add any other files needed by the application.
   a. Right-click the **Files** icon.
   b. Select **Add Files** from the pop-up menu.
   c. Select **Browse** to navigate the directory structure.
   d. Click **Select** to open an archive.
   e. Select the files to add and click **Add**.
   f. In the Selected Files window, click **OK** to add the files.

Assemble any other new modules of your choice:
- EJB modules
- Application client modules
- Resource adapter modules

You can also migrate existing modules.

Another option is to proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

Container transactions

Container transaction properties specify how an EJB container is to manage transaction scopes for the enterprise bean's method invocations. A transaction attribute is mapped to one or more methods.

Method extensions

Method extensions are IBM extensions to the standard deployment descriptors for enterprise beans.

Method extension properties are used to define transaction isolation levels for methods, to control the delegation of a principal's credentials, and to define custom finder methods.

Method permissions

A method permission is a mapping between one or more security roles and one or more methods that a member of the role can call.

References

References are logical names used to locate external resources for enterprise applications. References are defined in the application's deployment descriptor file. At deployment, the references are bound to the physical location (global JNDI name) of the resource in the target operational environment.

This product supports the following types of references:

- An EJB reference is a logical name used to locate the home interface of an enterprise bean.
- A resource reference is a logical name used to locate a connection factory object.

These objects define connections to external resources such as databases and messaging systems. The container makes references available in a JNDI naming subcontext. By convention, references are organized as follows:

- EJB references are made available in the java:comp/env/ejb subcontext.
- Resource references are made available as follows:
  - JDBC DataSource references are declared in the java:comp/env/jdbc subcontext.
  - JMS connection factories are declared in the java:comp/env/jms subcontext.
  - JavaMail connection factories are declared in the java:comp/env/mail subcontext.
  - URL connection factories are declared in the java:comp/env/url subcontext.

CMP field assembly settings

In Enterprise JavaBeans (EJB) Version 1.1-compliant beans, container-managed persistence (CMP) fields define the variables in the bean class for which the container must handle persistence management. In EJB Version 2.0-compliant beans, these are replaced by abstract get and set methods; generated code provides the implementation of these abstract methods.

Name

Specifies a subset of public variables in the enterprise bean's implementation class.

Container transaction assembly settings

Container transaction settings specify how an EJB container is to manage transaction scopes for the enterprise bean's method invocations. Specify one or more methods and associate a transaction attribute with each method.

Name

Specifies a name for the mapping between a transaction attribute and one or more methods.

Description

Contains text that describes the mapping.

Transaction attribute

Specifies how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean's business method.

Data type	String
Default	`Required`
Range	For all but message-driven beans, valid values are `Mandatory`, `Never`, `Not Supported`, `Required`, `Requires New`, `Supports`. For session beans, `Bean Managed` is also valid. For message-driven beans, only `Bean Managed`, `Not Supported`, and `Required` are valid.

Additional information about valid values follows:

Bean Managed
Notifies the container that the bean class directly handles transaction demarcation. This setting can be specified for session beans and (in EJB 2.0 implementations only) for message-driven beans, and it cannot be specified for individual bean methods.

Mandatory
Directs the container to always call the bean method within the transaction context associated with the client. If the client attempts to invoke the bean method without a transaction context, the container throws the javax.jts.TransactionRequiredException exception to the client. The transaction context is passed to any EJB object or resource accessed by an enterprise bean method.

EJB clients that access these entity beans must do so within an existing transaction. For other enterprise beans, the enterprise bean or bean method must implement the `Bean Managed` value or use the `Required` or `Requires New` value. For non-enterprise bean EJB clients, the client must access a transaction by using the javax.transaction.UserTransaction interface.

Never
Directs the container to invoke bean methods without a transaction context.
- If the client calls a bean method from within a transaction context, the container throws the java.rmi.RemoteException exception.
- If the client calls a bean method from outside a transaction context, the container behaves in the same way as if the Not Supported transaction attribute was set. The client must call the method without a transaction context.

Not Supported
Directs the container to call the bean method without a transaction context. If a client calls a bean method from within a transaction context, the container suspends the association between the transaction and the current thread before invoking the method on the enterprise bean instance. The container then resumes the suspended association when the method invocation returns. The suspended transaction context is not passed to any enterprise bean objects or resources that are used by this bean method.

Required
Directs the container to call the bean method within a transaction context.

If a client calls a bean method from within a transaction context, the container calls the bean method within the client transaction context. If a client calls a bean method outside a transaction context, the container creates a new transaction context and calls the bean method from within that context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

Requires New
Directs the container to always call the bean method within a new transaction context, regardless of whether the client calls the method within or outside a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

Supports
Directs the container to call the bean method within a transaction context if the client calls the bean method within a transaction. If the client calls the bean method without a transaction context, the container calls the bean method without a transaction context. The transaction context is passed to any enterprise bean objects or resources that are used by this bean method.

Methods - Name

Specifies the name of an enterprise bean method, or the asterisk character (*). The asterisk is used to denote all methods of an enterprise bean's remote and home interfaces.

Methods - Enterprise bean

Specifies which enterprise bean contains the methods indicated in the **Name** setting.

Methods - Type

Used to distinguish between a method with the same signature that is defined in both the home and remote interface. Use `Unspecified` if a transaction attribute applies to all methods of the bean.

Data type	String
Range	Valid values for EJB 1.1 implementations are `Home`, `Remote`, or `Unspecified`. For EJB 2.0 implementations, `Local` and `LocalHome` are also valid.

Methods - Parameters

Contains a list of fully qualified Java type names of the method parameters. This setting is used to identify a single method among multiple methods with an overloaded method name.

EJB module assembly settings

An EJB module is used to assemble enterprise beans into a single deployable unit. An EJB module contains one or more enterprise beans and a deployment descriptor.

File name

Specifies the file name of the EJB module, relative to the top level of the application package.

Alternate DD

Specifies a deployment descriptor to be used at run time instead of the one installed in the module.

Classpath

The path that contains additional classes required by the application that are not contained in the module's archive file. The class loader uses this path. Specify the values relative to the root of the EAR file and separate the values with spaces. Absolute values that refer to files or directories on the hard drive are ignored.
To specify classes that are not in JAR files but are in the root of the EAR file, use a period and forward slash (**./**). Consider the following example directory structure in which the file myapp.ear contains an EJB module named myejb.jar. Additional classes reside in class1.jar and class2.zip. A class named xyz.class is not packaged in a JAR file but is in the root of the EAR file.

```
myapp.ear/myejb.jar
myapp.ear/class1.jar
myapp.ear/class2.zip
myapp.ear/xyz.class
```

Specify `class1.jar class2.zip ./` as the value of the Classpath setting. (Name only the directory for .class files.)

Display name

Specifies a short name that is intended to be displayed by GUIs.

Description

Contains text that describes the module.

EJB client JAR

Specifies the location of a JAR file that contains a subset of deployed classes needed by the client.

Small icon

Specifies the name of a JPEG or GIF file that contains a small image (16x16 pixels). The image is used as an icon to represent the module in a GUI.

Large icon

Specifies the name of a JPEG or GIF file that contains a large image (32x32 pixels). The image is used as an icon to represent the module in a GUI.

Generalizations - Subtype

Information about this property is not available.
This property is an IBM extension to the standard J2EE deployment descriptor.

Generalizations - Supertype

Information about this property is not available.
This property is an IBM extension to the standard J2EE deployment descriptor.

EJB relationships - Name

The logical name for a container-managed relationship between EJB 2.0-compliant entity beans.

Default data source - JNDI name

Specifies the default JNDI name for the data source. This default is used if binding information is not specified in the deployment descriptor for an individual enterprise bean.

Default CMP connection factory

Specifies the JNDI name for a CMP connection factory. This setting is applicable only for EJB 2.x-compliant CMP beans.

Default authorization - User ID

Specifies the default user ID for connecting to an enterprise bean's data store.

Default authorization - Password

Specifies the default password for connecting to an enterprise bean's data store.

Entity bean assembly settings

An entity bean encapsulates persistent data, which is stored in a data source, and associated methods to manipulate that data.

EJB name

Specifies a logical name for the enterprise bean. This name must be unique within the EJB module. There is no relationship between this name and the JNDI name.

Display name

Specifies a short name that is intended to be displayed by GUIs.

Description

Contains text that describes the entity bean.

EJB class

Specifies the full name of the enterprise bean class (for example, com.ibm.ejs.doc.account.AccountBean).

Remote - Home

(Required for EJB 1.x) Specifies the full name of the enterprise bean's home interface class (for example, com.ibm.ejs.doc.account.AccountHome).

Remote - Interface

(Required for EJB 1.x) Specifies the full name of the enterprise bean's remote interface class (for example, com.ibm.ejs.doc.account.Account).

Local interface - Home

(Required for EJB 1.x) Specifies the full name of the enterprise bean's local home interface class (for example, com.ibm.ejs.doc.account.AccountLocalHome).

Local interface - Interface

(Required for EJB 1.x) Specifies the full name of the enterprise bean's local interface class (for example, com.ibm.ejs.doc.account.AccountLocal).

Persistence type

Specifies whether an entity bean manages its own persistent storage or whether storage is managed by the container.

Data type	String
Range	Valid values are `Bean managed` and `Container managed`.

Reentrant

Specifies whether the entity bean is reentrant. If an enterprise bean is reentrant, it can call methods on itself or call another bean that calls a method on the calling bean. Only entity beans can be reentrant.
If an entity bean is not reentrant and a bean instance is executing a client request in a transaction context and another client using the same transaction context makes a request on the same bean instance, the EJB container throws the java.rmi.RemoteException exception to the second client. If a bean is reentrant, the container cannot distinguish this type of illegal loopback call from a legal concurrent call, so the bean must be coded to detect illegal loopback calls.

Primary key class

Specifies the full name of the bean's primary key class (for example, com.ibm.ejs.doc.account.AccountKey). Composite primary keys map to multiple fields in the entity bean class (or to data structures built from the primitive Java data types) and must be encapsulated in a primary key class.
More complicated enterprise beans are likely to have composite primary keys, with multiple instance variables representing the primary key. A subset of the container-managed fields is used to define the primary key class associated with each instance of an enterprise bean.

Primary key field

Specifies the name of a simple primary key. Simple primary keys map to a single field in the entity bean class and are made up of primitive Java data types (such as integer or long). If exactly one CMP field is the primary key, it can be specified here.

Data type	String
Range	Valid values are the name of any one CMP field or `Compound key`, which appears when the primary key class is set

Version

Specifies the version of EJB specification with which a container-managed persistence (CMP) entity bean complies.

Data type	String
Range	Valid values are `1.x` or`2.x`

Abstract schema name

Specifies the name of the abstract schema type of an EJB Version 2.x CMP entity bean. It is used in EJB Query Language (QL) queries.
For example, the abstract schema name might be Order for an entity bean whose local interface is com.acme.commerce.Order.

Small icon

Specifies the name of a JPEG or GIF file that contains a small image (16x16 pixels). The image is used as an icon to represent the entity bean in a GUI.

Large icon

Specifies the name of a JPEG or GIF file that contains a large image (32x32 pixels). The image is used as an icon to represent the entity bean in a GUI.

Security identity

Specifies that a principal's credential properties are to be handled as indicated in the Run-As mode property. If this setting is enabled, the Run-As mode property can be edited.

Run-As mode

Specifies the credential information to be used by the security service to determine the permissions that a principal has on various resources.
At appropriate points, the security service determines whether the principal is authorized to use a particular resource based on the principal's permissions. If the method call is authorized, the security service acts on the principal's credential properties according to the Run-As mode setting of the enterprise bean.

Data type	Enumerated integer

Range	Valid values are `Use identity of caller` and `Use identity assigned to specified role`

Additional information about valid settings follows:

Use identity of caller
: The security service makes no changes to the principal's credential properties.

Use identity assigned to specified role
: A principal that has been assigned to the specified security role is used for the execution of the bean's methods. This association is part of the application binding in which the role is associated with a user ID and password of a user who is granted that role.

Role name

Specifies the name of a security role. If **Run-As mode** is set to `Use identity assigned to specified role`, a principal that has been granted this role is used.

Description

Contains further information about the security role.

Concurrency control

Specifies how the bean is to handle concurrent access to its data. This setting is applicable only for EJB 1.x-compliant entity beans.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Range	Valid values are `Optimistic` or `Pessimistic`

Inheritance root

Specifies whether the enterprise bean is at the root of an inheritance hierarchy.
This property is an IBM extension to the standard J2EE deployment descriptor.

Bean Cache - Activate at

Specifies the point at which an enterprise bean is activated and placed in the cache. Removal from the cache and passivation is also governed by this setting.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Default	`Transaction`
Range	Valid values are `Once`, `Transaction`, and `Activity session`

More information about valid values follows:

Once Indicates that the bean activates when it is first accessed in the server process, and passivates (and is removed from the cache) at the discretion of the container, for example, when the cache becomes full.

Transaction
: Indicates that the bean activates at the start of a transaction and passivates (and is removed from the cache) at the end of the transaction.

Activity session
: Indicates that the bean activates and passivates as follows:
  - On an ActivitySession boundary, if an ActivitySession context is present on activation
  - On a transaction boundary, if a transaction context (but no ActivitySession context) is present on activation

- Otherwise, on an invocation boundary

The values of the **Activate at** and **Load at** settings govern which commit options are used, as follows:

- For commit option A (implies exclusive database access), use `Activate at = Once` and `Load at = Activation`.

  This option reduces database I/O (avoids calls to the ejbLoad function) but serializes all transactions accessing the bean instance. Option A can increase memory usage by maintaining more objects in the cache, but can provide better response time if bean instances are not generally accessed concurrently by multiple transactions. To use Option A successfully, you must also set Concurrency control to `Pessimistic`.

  Note for Network Deployment users

  Note: When workload management is enabled, you cannot use Option A. You must use settings that result in the use of options B or C.
- For commit option B (implies shared database access), use `Activate at = Once` and `Load at = Transaction`.

  Option B can increase memory usage by maintaining more objects in the cache. However, because each transaction creates its own copy of an object, there can be multiple copies of an instance in memory at any given time (one per transaction), requiring database access at each transaction. If an enterprise bean contains a significant number of calls to the ejbActivate function, using Option B is beneficial because the required object is already in the cache. Otherwise, this option does not provide significant benefits over Option A.
- For commit option C (implies shared database access), use `Activate at = Transaction` and `Load at = Transaction`.

  This option reduces memory usage by maintaining fewer objects in the cache; however, there can be multiple copies of an instance in memory at any given time (one per transaction). This option can reduce transaction contention for enterprise bean instances that are accessed concurrently but not updated.

Bean Cache - Load at

Specifies when the bean loads its state from the database. The value of this setting implies whether the container has exclusive or shared access to the database.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Default	`Transaction`
Range	Valid values are `Activation` and `Transaction`

Additional information about valid values follows:

Activation
: Indicates that the bean loads when it is activated (regardless of Activate at setting) and implies that the container has exclusive access to the database.

Transaction
: Indicates that the bean loads at the start of a transaction and implies that the container has shared access to the database.

The Activate at and Load at settings govern which commit options are used. The commit options are described in the Enterprise JavaBeans specification. For more information about this setting and achieving a given commit behavior, see Bean Cache - Activate at.

Commit option

Specifies which commit option is used as a result of bean cache settings. The commit options are described in the Enterprise JavaBeans specification.

Data type	String
Range	Valid values are `A`, `B`, and `C`

Local Transactions - Unresolved action

Specifies the action that the EJB container must take if resources are uncommitted by an application in a local transaction.
This property is an IBM extension to the standard J2EE deployment descriptor. This setting is applicable only when **Resolution control** is set to `Application`. A local transaction context is created when a method runs in what the EJB specification refers to as an unspecified transaction context.

Data type	String
Default	`Rollback`
Range	Valid values are `Commit` and `Rollback`

Additional information about these settings follows:

Commit
: At end of the local transaction context, the container instructs all unresolved local transactions to commit.

Rollback
: (Default) At end of the local transaction context, the container instructs all unresolved local transactions to roll back.

Local Transactions - Resolution control

Specifies how the local transaction is to be resolved before the local transaction context ends: by the application through user code or by the EJB container.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Range	Valid values are `Application` and `ContainerAtBoundary`

Additional information about these settings follows:

Application
: When this setting is used, your code must either commit or roll back the local transaction. If this does not occur, the runtime environment logs a warning and automatically commits or rolls back the connection as specified by the **Unresolved action** setting.

ContainerAtBoundary
: When this setting is used, the container takes responsibility for resolving each local transaction. This provides you with a programming model similar to global transactions in which your code simply gets a connection and performs work within it. User code does not have to handle local transactions.

- If the **Boundary** attribute is set to ActivitySession, then the local transactions are enlisted as ActivitySession resources and directed to complete by the ActivitySession.
- If the the **Boundary** attribute is set to BeanMethod, then the local transactions are committed at method end by the container.

Connections are never committed automatically by the resource adapter when this value is configured for the bean **Unresolved action** is not used. An application cannot call Connection.LocalTransaction.begin() when using this policy and receives an exception from the resource adapter if it does so.

When using a **Resolution control** of `ContainerAtBoundary`, applications must get connection handles *after* the local transaction context boundary has been started by the container. The application should close the connection before the end of the boundary, although any work performed on the connection is not committed or rolled back until the local transaction context ends. This model of connection usage is sometimes referred to as the "get-use-close" model.

This value is supported only for EJB components that use container-managed transactions. It is not supported for web components or for enterprise beans that use bean-managed transactions.

Local Transactions - Boundary

Specifies the duration of a local transaction context.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Default	`BeanMethod`
Range	Valid values are `BeanMethod` and `ActivitySession`

Additional information about valid settings follows:

BeanMethod
When this setting is used, the local transaction begins when the method begins and ends when the method ends.

ActivitySession
When this setting is used, the local transaction must be resolved within the scope of any ActivitySession in which it was started or, if no ActivitySession context is present, within the same bean method in which it was started.

This property can be changed on WebSphere Application Server Enterprise only.

Local Relationship Roles - Name

Within a local relationship between EJB 1.x-compliant entity beans, the logical name for the view an entity bean presents to other beans in the relationship. For example, in a relationship between Account and Customer beans, the role of the Account instance relative to the Customer instance might be *savingsAccount*.

This property is an IBM extension to the standard J2EE deployment descriptor. This is separate from the container-managed relationships defined in the Enterprise JavaBeans specification, Version 2.0.

Local Relationship Roles - Source EJB Name

The name of the entity bean for which the role is defined.
This property is an IBM extension to the standard J2EE deployment descriptor.

Local Relationship Roles - is Forward

Specifies how deployment code for navigating the relationship is generated. This setting is applicable only for navigable relationships.

If **isForward** is enabled (set to `true`), deployment code is generated in the source bean. That is, navigation of the relationship proceeds *forward* from the source to the target.

Otherwise, deployment code is generated in the target bean. That is, navigation of the relationship proceeds from the target to the source.

This property is an IBM extension to the standard J2EE deployment descriptor. For more information, see the documentation for the Deployment Tool for Enterprise JavaBeans.

Local Relationship Roles - is Navigable

Specifies whether data in related beans may be retrieved through queries to the source bean.
This property is an IBM extension to the standard J2EE deployment descriptor.

Lifetime in cache

The lifetime, in seconds, of cached data for an instance of this bean type.
This value indicates how long the cached data is to exist beyond the end of the transaction in which the data was retrieved. This might avoid another retrieval from persistent storage if the same bean instance were to be used in later transactions. How this value is interpreted depends on the value of **Lifetime in cache usage**.

This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	Long
Units	Seconds
Default	0
Range	0 to 2^{61} - 1

Lifetime in cache usage

Indicates how the lifetime-in-cache setting is to be used by the caching mechanism.
This property is an IBM extension to the standard J2EE deployment descriptor.

If your application uses CMP beans in which the underlying data changes infrequently, you might gain significantly better performance by using this setting with **Lifetime in cache**. Typically, data read from persistent storage is held temporarily in an internal cache until the state of the instance is restored. Cached data normally does not persist beyond state restoration or the end of the transaction in which the finder method was called. By setting **Lifetime in cache usage** to a value other than `Off`, you indicate that the cached data is to be held for a longer time, potentially hours or days, before invalidating the version of the data in the cache and fetching a new version. Avoiding a trip to persistent storage greatly speeds up access to such beans by applications.

In addition, the use of a value other than `Off` requires that finders on the bean have an access type of Optimistic Read (if you are only reading instances of the bean) or Optimistic Update (if you plan to occasionally update instances of the bean).
- For EJB 1.x-compliant beans, see Access intent - access type.
- For EJB 2.x-compliant beans, see Applied access intent.

Setting Load at to `activation` and Activate at to `Once` also minimizes retrievals from persistent storage. However, this settings combination might not be supported by certain CMP beans because it results in the ejbLoad() method being

called once instead of at the beginning of each transaction in which they are used. The lifetime-in-cache settings combination is independent of CMP bean implementation, though it does incur the modest overhead of calling ejbLoad() on each use.

Data type	Enumerated int
Units	Not applicable
Default	0 (`Off`)
Range	Valid values are `Clock Time`, `Elapsed Time`, `Week Time`, or `Off`

Additional information about valid values follows:

Off
When this value is used, the value of **Lifetime in cache** is ignored. Beans of this type are cached only in a transaction-scoped cache. The cached data for this instance expires after the transaction in which it was retrieved is completed.

Elapsed Time
When this value is used, the value of **Lifetime in cache** is added to the time at which the transaction in which the bean instance was retrieved is completed. The resulting value becomes the time at which the cached data expires. The value of **Lifetime in cache** can add up to minutes, hours, days, and so on.

Clock Time
When this value is used, the value of **Lifetime in cache** represents a particular time of day. The value is added to the immediately preceeding or following midnight to calculate a future time value, which is then treated as for `Elapsed Time`. Using `Clock Time` enables you to specify that all instances of this bean type are to have their cached data invalidated at, for example, 3 AM, no matter when they were retrieved. This is important if, for example, the data underlying this bean type is batch-updated at 3 AM every day.

The selection of midnight (preceding or following) depends on the value of **Lifetime in cache**. If **Lifetime in cache** plus the value that represents the preceeding midnight is earlier than the current time, the following midnight is used.

When you use `Clock Time`, the value of **Lifetime in cache** is not supposed to represent more than 24 hours. If it does, the cache manager subtracts 24-hour increments from it until a value less than or equal to 24 hours is achieved. To invalidate data at midnight, set **Lifetime in cache** to 0.

Week Time
Usage of this value is the same as for `Clock Time`, except that the value of **Lifetime in cache** is added to the preceeding or following Sunday midnight (11:59 PM Saturday plus 1 minute). When `Week Time` is used, the value of **Lifetime in cache** can represent more than 24 hours but not more than 7 days.

Default Access Intent

Specifies the default access intent under which the entity should load.

5.0.2 +

Data type	String
Units	Not applicable
Default	Not applicable

Range	Valid settings are wsPessimisticUpdate, wsPessimisticUpdate-NoCollision, wsPessimisticUpdate-Exclusive, wsPessimisticUpdate-WeakestLockAtLoad, wsPessimisticRead, wsOptimisticUpdate, or wsOptimisticRead.

JNDI name

Specifies the JNDI name of the bean's home interface. This is the name under which the enterprise bean's home interface is registered and therefore, is the name that must be specified when an EJB client does a lookup of the home interface.

Data source - JNDI name

Specifies the JNDI name for the bean's data source.

Default Authorization - User ID

Specifies the default user ID for connecting to a data source.

Default Authorization - Password

Specifies the default password for connecting to a data source.

CMP Resource - JNDI name

Specifies the JNDI name for the resource by which CMP data is stored.

CMP Resource - Resource authentication

Specifies the scope at which resources are to be authenticated: by the container or by the resource.

Message-driven bean assembly settings

Use this page to configure the assembly properties of message-driven beans

For more information about the effect of JMS properties, such as message selectors and message acknowledgement, see the WebSphere MQ *Using Java* book, SC34-5456 or Sun's Java Message Service (JMS) specification documentation

The following notebook pages are available:

Page tab
: **Description**

General properties
: Specify general assembly properties for the message bean.
- EJB name
- Display name
- Description
- EJB class
- Transaction type

Advanced properties
: Specify advanced assembly properties for the message bean.
- Message selector
- Acknowledge mode
- Destination type

Bindings properties
: Specify bindings assembly properties for the message bean.
- Listener port name

EJB name

The logical name for the message bean (as an enterprise bean)

The logical name for the message bean (as an enterprise bean). This name must be unique within the EJB module. There is no relationship between this name and the JNDI name.

Data type	String

Display name

A short name that is intended to be displayed by graphical user interfaces

Data type	String

Description

A description of the message bean, for administrive use

Data type	String

EJB class

The full package name of the message bean class

Specify the full package name of the message bean class, for example, `com.ibm.ejs.doc.account.MessageBean`. You can either type the class name or click **Browse** to locate an existing class file.

Data type	String

Transaction type

Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean

Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message bean.

Data type	Enum
Default	Bean
Range	**Bean** The message bean manages its own transactions **Container** The container manages transactions on behalf of the bean

Message selector

The JMS message selector to be used to determine which messages the message bean receives

The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

Data type	String
Range	A String whose syntax is based on a subset of the SQL92 conditional syntax.

Acknowledge mode

How the session acknowledges any messages it receives.
This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

Data type
Enum

Default
Auto Acknowledge

Range

Auto Acknowledge
The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

Dups OK Acknowledge
The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

Destination type

Whether the message bean uses a queue or topic destination.

Data type
Enum

Default
Null

Range

Queue The message bean uses a queue destination.

Topic The message bean uses a topic destination.

Listener port name

The name of the listener port for this message bean.
The name of the listener port for this message bean (as defined on the WebSphere administrative console).

Data type
String

Method extension assembly settings

Method extensions are IBM extensions to the standard J2EE deployment descriptors for Enterprise JavaBeans (EJB) Version 1.x-compliant beans. Method extension settings define transaction isolation levels for methods and control the delegation of a principal's credentials.

Method type

Specifies the type of the enterprise bean method.

Data type
String

Range
Valid values are `Home`, `Remote`, and `Unspecified`.

Name

Specifies the name of an enterprise bean method, or the asterisk character (*). The asterisk is used to denote all methods of an enterprise bean's remote and home interfaces.

Parameters

Contains a list of fully qualified Java type names of the method parameters. Used to identify a single method among multiple methods with an overloaded method name.

Isolation level attributes

The transaction isolation level determines how isolated one transaction is from another. This can be set for individual methods in an enterprise bean or for all methods in the enterprise bean. An asterisk is used to indicate all methods in the bean. This setting is not applicable for EJB 2.x-compliant beans.

Within a transactional context, the isolation level associated with the first method call becomes the required isolation level for all methods called within that transaction. If a method is called with a different isolation level from that of the first method, the java.rmi.RemoteException exception is thrown.

Isolation level

Specifies the level of transactional isolation.

The container uses the transaction isolation level attribute as follows:

- Session beans and entity beans with bean-managed persistence (BMP): For each database connection used by the bean, the container sets the transaction isolation level at the start of each transaction unless the bean explicitly sets the isolation level on the connection.
- Entity beans with container-managed persistence (CMP): The container generates database access code that implements the specified isolation level.

Data type	String
Range	Valid values are `Serializable`, `Repeatable read`, `Read committed`, and `Read uncommitted`

Serializable

This level prohibits the following types of reads:

- *Dirty* reads, in which a transaction reads a database row containing uncommitted changes from a second transaction.
- *Nonrepeatable* reads, in which one transaction reads a row, a second transaction changes the same row, and the first transaction rereads the row and gets a different value.
- *Phantom* reads, in which one transaction reads all rows that satisfy an SQL WHERE condition, a second transaction inserts a row that also satisfies the WHERE condition, and the first transaction applies the same WHERE condition and gets the row inserted by the second transaction.

Repeatable read

This level prohibits dirty reads and nonrepeatable reads, but it allows phantom reads.

Read committed

This level prohibits dirty reads but allows nonrepeatable reads and phantom reads.

Read uncommitted

This level allows dirty reads, nonrepeatable reads, and phantom reads.

Access intent - Intent type

Specifies whether to load the enterprise bean as read-only or for update. This setting is applicable only for EJB 1.x-compliant beans.

This setting is applicable for the following types of beans:
- EJB 1.x-compliant entity beans
- Enterprise beans with CMP version 1.x that are packaged in EJB 2.x-compliant modules

To specify the access intent for EJB 2.x-compliant beans, select an access intent policy.

Data type	String
Range	Valid values are `Read` or `Update`

Finder descriptor - User

Specifies that the user has provided a finder helper class in the entity bean's home interface. The class contains specialized finder methods. This setting is applicable only for EJB 1.x-compliant entity beans.

Finder descriptor - EJB QL

Describes the semantics of a finder method that uses EJB QL (Enterprise JavaBeans query language). This setting is applicable only for EJB 1.x-compliant entity beans. EJB QL is a declarative, SQL-like language that is intended to be compiled to the target language of the persistent datastore used by a persistence manager. The language is independent of the bean's mapping to a relational datastore and is therefore portable. The EJB query specifies a search based on the persistent attributes and relationships of the bean. An EJB query can contain the following clauses:
- SELECT (optional), which specifies the EJB objects to return
- FROM (required), which specifies the collections of objects to which the query is to be applied
- WHERE (optional), which contains search predicates over the collections
- ORDER BY (optional), which specifies the ordering of the resulting collection

Finder descriptor - Full SELECT

Describes the semantics of a finder method that uses an SQL SELECT clause. For information on restrictions, see the documentation for the Deployment Tool for Enterprise JavaBeans.

Finder descriptor - WHERE clause

Describes the semantics of a finder method that uses an SQL WHERE clause. This clause restricts the results that are returned by the query. For information on restrictions, see the documentation for the Deployment Tool for Enterprise JavaBeans.

Security identity

Specifies whether a principal's credential settings are to be handled as indicated in the **Run-As mode** setting. If this is enabled, the **Run-As mode** setting can be edited.

Description

Contains further information about the security instructions.

Run-As mode

Specifies the credential information to be used by the security service to determine the permissions that a principal has on various resources.
At appropriate points, the security service determines whether the principal is authorized to use a particular resource based on the principal's permissions. If the method call is authorized, the security service acts on the principal's credential settings according to the **Run-As mode** setting of the enterprise bean.

Data type	Enumerated integer
Range	Valid values are `Use identity of caller`, `Use identity of EJB server`, and `Use identity assigned to specified role`

Additional information about valid values for this setting follows:

Use identity of caller
: The security service makes no changes to the principal's credential settings.

Use identity of EJB server
: The security service alters the principal's credential settings to match the credential settings associated with the EJB server.

Use identity assigned to specified role
: A principal that has been assigned to the specified security role is used for the execution of the bean's methods. This association is part of the application binding in which the role is associated with a user ID and password of a user who is granted that role.

Role name

Specifies the name of a security role. If **Run-As mode** is set to `Use identity assigned to specified role`, a principal that has been granted this role is used.

Description

Contains further information about the security role.

Method permission assembly settings

A method permission is a mapping between one or more security roles and one or more methods that a member of the role can call. Assembly settings for method permissions include an optional description, a list of security role names, and a list of methods. The security roles must be defined, and the methods must be defined in the enterprise bean's remote or home interfaces.

Method permission name

Specifies a name for the mapping between method permissions and security roles.

Description

Contains text that describes the mapping between method permissions and security roles.

Methods - Name

Specifies the name of an enterprise bean method, or the asterisk (*) character. The asterisk is used to denote all the methods of an enterprise bean's remote and home interfaces.

Methods - Enterprise bean

Specifies the name of the enterprise bean that contains the method.

Methods - Type

Distinguishes between a method with the same signature that is defined in both the home and remote interface. Use `Unspecified` if a method permission applies to all methods of a bean.

Data type	String
Range	Valid values are `Unspecified`, `Remote`, or `Home`.

Methods - Parameters

Contains a list of fully qualified Java type names of the method parameters. This setting is used to identify a single method among multiple methods with an overloaded method name.

Unchecked

Specifies whether the method permission is checked before the method is run.

Roles - Role name

Specifies the name of the security role that must be granted in order to call the method.

Query assembly settings

Use these to specify a finder or SELECT query.

A query element contains the following:
- Optional description of the query
- Name of the finder or SELECT method that uses the query
- The return type of mapping, if it is used
- Whether the query is for a SELECT method
- EJB query language (EJB QL) query string that defines the query

Queries that are expressed in EJB QL must use the ejb-ql element to specify the query. If a query cannot be expressed in EJB QL, describe the semantics of the query by using the description element and leave the ejb-ql element empty.

Name

Contains the name of an enterprise bean method or the asterisk (*) character. An asterisk in the method-name element denotes all methods of an enterprise bean's remote and home interfaces.

Parameters

Contains a list of the fully-qualified Java names of the method parameters.

Result type

Used in the query element to indicate whether a returned abstract schema type for a SELECT method should be mapped to an EJBLocalObject or EJBObject type.

EJB reference assembly settings

An EJB reference is a logical name used to locate the home interface of an enterprise bean used by an application.

At deployment, the EJB reference is bound to the enterprise bean's home in the target operational environment. The container makes the application's EJB references available in a JNDI naming context. It is recommended that references to enterprise beans be organized in the ejb subcontext of the application's environment (in `java:comp/env/ejb`).

Name

Specifies the JNDI name of the enterprise bean's home interface relative to the `java:comp/env` context.
For example, if ejb/EmplRecord is specified, the referring code looks up the enterprise bean's home interface at java:comp/env/ejb/EmplRecord. This JNDI name is an alias used by the code (the actual JNDI name is specified on the Binding tab).

Description

Contains text that describes the EJB reference.

Link

Used to link an EJB reference to an enterprise bean in the current module (the same module as the one making the reference) or in another module within the same J2EE application. This setting specifies the name of the target enterprise bean. The target enterprise bean can be in any EJB module in the same J2EE application as the referring module. To avoid having to rename enterprise beans to have unique names within an J2EE application, specify the path name of the EJB archive file that contains the referenced enterprise bean and append the target bean's name, separated by a # symbol (for example, `...products/product.jar#ProductEJB`). The path name is relative to the referring module's archive file specification. If a link is not specified, the reference must be resolved to a JNDI name during installation.

Home

Specifies the fully qualified name of the enterprise bean's home interface (for example, com.ibm.ejbs.EmplRecordHome).

Remote

Specifies the fully qualified name of the enterprise bean's remote interface (for example, com.ibm.ejbs.EmplRecord).

Type

Specifies the expected type of the referenced enterprise bean.

Data type	String
Default	None; must be set
Range	`Entity` or `Session`

JNDI name

Binding information that is used by the run-time environment to resolve the location of a resource.

For EJB references, the value of this setting must match the JNDI name of the enterprise bean as it was specified on the **Binding** tab for the EJB module that contains the bean.

EJB local-reference assembly settings

For EJB 2.0-compliant beans, the EJB local reference element declares a reference to another enterprise bean's local home interface.

Name

Specifies the name of an EJB reference.

This is the JNDI name that the servlet code uses to get a reference to the enterprise bean. The following example illustrates how this element is specified in the deployment descriptor:

```
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
```

Description

Contains a description of the parent element.

This can include any information that the EJB archive-file producer wants to provide to the consumer of the EJB archive file.

Link

Used in the ejb-ref element to specify that an EJB reference is linked to an enterprise bean in the encompassing web-application package.

The value of the link element must be the EJB name of an enterprise bean in the same web-application package. The following example illustrates how this element is specified in the deployment descriptor:

```
<ejb-link>EmployeeRecord</ejb-link>
```

Local interface

Specifies the fully-qualified name of the enterprise bean's local interface.

Local home

Specifies the fully-qualified name of the enterprise bean's local home interface.

Type

Specifies the expected type of the referenced enterprise bean.

EJB relation assembly settings

An EJB relation describes a relationship between two entity beans with container-managed persistence.

The name of the relationship, if specified, is unique within an EJB archive file.

Description

Contains text to describe the EJB relationship role.

Source EJB

Specifies the source of the role that participates in a relationship.

Multiplicity

Specifies the multiplicity of the role that participates in a relation.

Cascade delete

Within a particular relationship, specifies that the lifetime of one or more entity beans is dependent on the lifetime of another entity bean.
Cascade delete can be specified only for an EJB relationship role contained in an EJB relation in which the other EJB relationship role specifies a multiplicity of one.

CMR field

Enables the declaration of a container-managed relationship (CMR) field.
The CMR field describes the bean provider's view of a relationship. It consists of an optional description and the name and class type of the source enterprise bean's role in a relationship.

Exclude list assembly settings

The exclude list indicates which methods in the enterprise beans may not be called. You should also configure security for the enterprise bean so that access to the listed methods is not permitted.

This capability applies only to Enterprise JavaBeans (EJB) Version 2.x-compliant beans. For more information about exclude lists, see the EJB specification.

Description

Provides additional information about this exclude list.

Methods - Name

Specifies the name of an enterprise bean method, or the asterisk (*) character. The asterisk is used to denote all the methods of an enterprise bean's remote and home interfaces.

Methods - Enterprise bean

Specifies the name of the enterprise bean that contains the method.

Methods - Type

Distinguishes between a method with the same signature that is defined in both the home and remote interface. Use Unspecified if the exclusion applies to all methods of a bean.

Data type	String
Range	Valid values are `Unspecified`, `Remote`, or `Home`

Methods - Parameters

Contains a list of fully qualified Java type names of the method parameters. This setting is used to identify a single method among multiple methods with an overloaded method name.

Security role assembly settings

A security role is a logical grouping of principals. Access to operations (such as enterprise-bean methods) is controlled by granting access to a role.

Role name

Specifies the name of a security role that is unique to an application. This setting applies only when you are specifying security roles at the application level (EAR file).

Description

Contains text that describes the application-specific security role. This setting applies only when you are specifying security roles at the application level (EAR file).

Binding - Groups - Name

Specifies the user groups that are granted the application-specific security role. This setting applies only when you are specifying security roles at the application level (EAR file).

Binding - Users - Name

Specifies the users that are granted the application-specific security role. This setting applies only when you are specifying security roles at the application level (EAR file).

Binding - Special Subjects - Name

Specifies one of two special categories of authenticate users to which application-specific security roles can be granted: `Everyone` or `All`. This setting applies only when you are specifying security roles at the application level (EAR file).
If the special subject `All` is granted a role, any user who can authenticate by using a valid user ID and password is considered to be granted that role.

If the special subject `Everyone` is granted a role, all users, including those who did not authenticate, are granted the role. In other words, a method on an enterprise bean or a URI is unprotected if any of the required roles for that method are granted to the special subject `Everyone`.

Data type	String
Range	Valid values are `All` or `Everyone`

Session bean assembly properties

A session bean encapsulates transient data that is associated with a particular EJB client. Unlike data in an entity bean, the data in a session bean is not stored in a persistent data source.

EJB name

Specifies a logical name for the enterprise bean. This name must be unique within the EJB module. There is no relationship between this name and the JNDI name.

Display name

Specifies a short name that is intended to be displayed by GUIs.

Description

Contains text that describes the session bean.

EJB class

Specifies the full name of the enterprise bean class (for example, com.ibm.ejs.doc.account.AccountBean).

Remote - Home

Specifies the full name of the enterprise bean's home interface class (for example, com.ibm.ejs.doc.account.AccountHome).

Remote - Interface

Specifies the full name of the enterprise bean's remote interface class (for example, com.ibm.ejs.doc.account.Account).

Local interface - Home

Specifies the full name of the enterprise bean's home interface class (for example, com.ibm.ejs.doc.account.AccountLocalHome).

Local interface - Interface

Specifies the full name of the enterprise bean's local interface class (for example, com.ibm.ejs.doc.account.AccountLocal).

Session type

Specifies whether the enterprise bean maintains a conversational state (is stateful) or does not (is stateless).

Data type	String
Range	Valid values are `Stateful` and `Stateless`

Transaction type

Specifies whether the enterprise bean manages its own transactions or whether the container manages transactions on behalf of the bean.

Data type	String
Range	Valid values are `Container` or `Bean`

Small icon

Specifies the name of a JPEG or GIF file that contains a small image (16x16 pixels). The image is used as an icon to represent the session bean in a GUI.

Large icon

Specifies the name of a JPEG or GIF file that contains a large image (32x32 pixels). The image is used as an icon to represent the session bean in a GUI.

Security identity

Specifies whether a principal's credential properties are to be handled as indicated in the **Run-As mode** property. If this setting is enabled (that is, set to `true`), the **Run-As mode** setting can be edited.

Description

Contains further information about the security instructions.

Run-As mode

Specifies the credential information to be used by the security service to determine the permissions that a principal has on various resources.

At appropriate points, the security service determines whether the principal is authorized to use a particular resource based on the principal's permissions. If the method call is authorized, the security service acts on the principal's credential properties according to the **Run-As mode** setting of the enterprise bean.

Data type	Enumerated integer
Range	Valid values are `Use identity of caller` and `Use identity assigned to specified role`

Additional information about valid values for this setting follows:

Use identity of caller
: The security service makes no changes to the principal's credential properties.

Use identity assigned to specified role
: A principal that has been assigned to the specified security role is used for the execution of the bean's methods. This association is part of the application binding in which the role is associated with a user ID and password of a user who is granted that role.

Role name

Specifies the name of a security role. If **Run-As mode** is set to `Use identity assigned to specified role`, a principal that has been granted this role is used.

Description

Contains further information about the security role.

Timeout

This property applies only to stateful session beans.

This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	Integer
Units	Seconds

Inheritance root

Specifies whether the enterprise bean is at the root of an inheritance hierarchy.

This property is an IBM extension to the standard J2EE deployment descriptor.

Bean Cache - Activate at

Specifies the point at which an enterprise bean is activated and placed in the cache. Removal from the cache and passivation is also governed by this setting. This setting applies to stateful session beans only (not to stateless beans).

This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Default	`Once`
Range	Valid values are `Once` and `Transaction`

Additional information about valid values follows:

Once Indicates that the bean is activated when it is first accessed in the server process, and passivated (and removed from the cache) at the discretion of the container, for example, when the cache becomes full.

Transaction
Indicates that the bean is activated at the start of a transaction and passivated (and removed from the cache) at the end of the transaction.

Local Transactions - Unresolved action

Specifies the action that the EJB container must take if resources are uncommitted by an application in a local transaction.

This property is an IBM extension to the standard J2EE deployment descriptor. This setting is applicable only when **Resolution control** is set to `Application`. A local transaction context is created when a method runs in what the EJB specification refers to as an unspecified transaction context.

Data type	String
Default	`Rollback`
Range	Valid values are `Commit` and `Rollback`

Additional information about these settings follows:

Commit
At end of the local transaction context, the container instructs all unresolved local transactions to commit.

Rollback
(Default) At end of the local transaction context, the container instructs all unresolved local transactions to roll back.

Local Transactions - Resolution control

Specifies how the local transaction is to be resolved before the local transaction context ends: by the application through user code or by the EJB container.

This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Range	Valid values are `Application` and `ContainerAtBoundary`

Additional information about these settings follows:

Application
When this setting is used, your code must either commit or roll back the local transaction. If this does not occur, the runtime environment logs a warning and automatically commits or rolls back the connection as specified by the **Unresolved action** setting.

ContainerAtBoundary
When this setting is used, the container takes responsibility for resolving each local transaction. This provides you with a programming model similar to global transactions in which your code simply gets a connection and performs work within it. User code does not have to handle local transactions.

- If the **Boundary** attribute is set to ActivitySession, then the local transactions are enlisted as ActivitySession resources and directed to complete by the ActivitySession.

- If the the **Boundary** attribute is set to BeanMethod, then the local transactions are committed at method end by the container.

Connections are never committed automatically by the resource adapter when this value is configured for the bean **Unresolved action** is not used. An application cannot call Connection.LocalTransaction.begin() when using this policy and receives an exception from the resource adapter if it does so.

When using a **Resolution control** of `ContainerAtBoundary`, applications must get connection handles *after* the local transaction context boundary has been started by the container. The application should close the connection before the end of the boundary, although any work performed on the connection is not committed or rolled back until the local transaction context ends. This model of connection usage is sometimes referred to as the “get-use-close” model.

This value is supported only for EJB components that use container-managed transactions. It is not supported for web components or for enterprise beans that use bean-managed transactions.

Local Transactions - Boundary

Specifies the duration of a local transaction context. This property does not apply to stateless session beans.
This property is an IBM extension to the standard J2EE deployment descriptor.

Data type	String
Default	`BeanMethod`
Range	Valid values are `BeanMethod` and `ActivitySession`

Additional information about valid settings follows:

BeanMethod
: When this setting is used, the local transaction begins when the method begins and ends when the method ends.

ActivitySession
: When this setting is used, the local transaction must be resolved within the scope of any ActivitySession in which it was started or, if no ActivitySession context is present, within the same bean method in which it was started.

This property can be changed on WAS Enterprise only.

JNDI name

Specifies the JNDI name of the bean’s home interface. This is the name under which the enterprise bean’s home interface is registered and therefore, is the name that must be specified when an EJB client does a lookup of the home interface.

EJB containers

An Enterprise JavaBeans (EJB) container provides a run-time environment for enterprise beans within the application server. The container handles all aspects of an enterprise bean’s operation within the application server and acts as an intermediary between the user-written business logic within the bean and the rest of the application server environment.

One or more EJB modules, each containing one or more enterprise beans, can be installed in a single container.

The EJB container provides many services to the enterprise bean, including the following:
- Beginning, committing, and rolling back transactions as necessary.
- Maintaining pools of enterprise bean instances ready for incoming requests and moving these instances between the inactive pools and an active state, ensuring that threading conditions within the bean are satisfied.
- Most importantly, automatically synchronizing data in an entity bean's instance variables with corresponding data items stored in persistent storage.

By dynamically maintaining a set of active bean instances and synchronizing bean state with persistent storage when beans are moved into and out of active state, the container makes it possible for an application to manage many more bean instances than could otherwise simultaneously be held in the application server's memory. In this respect, an EJB container provides services similar to virtual memory within an operating system.

Between transactions, the state of an entity bean can be cached. The EJB container supports option A, B, and C caching.
- With option A caching, the application server assumes that the entity bean is used within a single container. Clients of that bean must direct their requests to the bean instance within that container. The entity bean has exclusive access to the underlying database, which means that the bean cannot be cloned or participate in workload management if option A caching is used.
- With option B caching, the entity bean remains active in the cache throughout the transaction but is reloaded at the start of each method call.
- With option C caching (the default), the entity bean is always reloaded from the database at the beginning of each transaction. A client can attempt to access the bean and start a new transaction on any container that has been configured to host that bean. This is similar to the session clustering facility described for HTTP sessions in that the entity bean's state is maintained in a shared database that can be accessed from any server when required.

This product supports the cloning of stateful session bean home objects among multiple application servers. However, it does not support the cloning of a specific instance of a stateful session bean. Each instance of a particular stateful session bean can exist in just one application server and can be accessed only by directing requests to that particular application server. State information for a stateful session bean cannot be maintained across multiple members of a server cluster.

For more information about EJB containers, see "Resources for learning."

Managing EJB containers

Each application server can have a single EJB container; one is created automatically for you when the application server is created. The following steps are to be performed only as needed to improve performance after the EJB application has been deployed.

1. Adjust EJB container settings.
2. Adjust EJB cache settings.

If adjustments do not improve performance, consider adjusting access intent policies for entity beans, reassembling the module, and redeploying the module in the application.

EJB container settings

Use this page to configure and manage a specific EJB container.

To view this administrative console page, click **Servers > Application Servers >** *serverName* **> EJB Container**.

Passivation directory

Specifies the directory into which the container saves the persistent state of passivated stateful session beans.

Beans are passivated when the number of active bean instances becomes greater than the cache size specified in the container configuration. When a stateful bean is passivated, the container serializes the bean instance to a file in the passivation directory and discards the instance from the bean cache. If, at a later time, a request arrives for the passivated bean instance, the container retrieves it from the passivation directory, deserializes it, returns it to the cache, and dispatches the request to it. If any step fails (for example, if the bean instance is no longer in the passivation directory), the method invocation fails.

Inactive pool cleanup interval

Specifies the interval at which the container examines the pools of available bean instances to determine if some instances can be deleted to reduce memory usage.

Data type	Integer
Units	Milliseconds
Range	0 to 2 147 483 674

Default datasource JNDI name

Specifies the JNDI name of a data source to use if no data source is specified during application deployment. This setting is not applicable for EJB 2.x-compliant CMP beans.

Servlets and enterprise beans use *data sources* to obtain these connections. When configuring a container, you can specify a default data source for the container. This data source becomes the default data source used by any entity beans installed in the container that use container-managed persistence (CMP).

The default data source for a container is secure. When specifying it, you must provide a user ID and password for accessing the data source.

Specifying a default data source is optional if each CMP entity bean in the container has a data source specified in its configuration. If a default data source is not specified and a CMP entity bean is installed in the container without specifying a data source for that bean, applications cannot use that CMP entity bean.

Initial state

Specifies the execution state requested when the server first starts.

Data type	String
Default	Started
Range	Valid values are `Started` and `Stopped`

EJB container system properties

In addition to the settings accessible from the administrative console, you can set the following system property by command-line scripting:

com.ibm.websphere.ejbcontainer.poolSize

Specifies the size of the pool for the specified bean type. This property applies to stateless, message-driven and entity beans. If you do not specify a default value, the container defaults of 50 and 500 are used.

Set the pool size for a given entity bean as follows:

```
beantype=min,max[:beantype=min,max...]
```

beantype is the J2EE name of the bean, formed by concatenating the application name, the # character, the module name, the # character, and the name of the bean (that is, the string assigned to the `<ejb-name>` field in the bean's deployment descriptor). *min* and *max* are the minimum and maximum pool sizes, respectively, for that bean type. Do not specify the square brackets shown in the previous prototype; they denote optional additional bean types that you can specify after the first. Each bean-type specification is delimited by a colon (:).

Use an asterisk (*) as the value of *beantype* to indicate that all bean types are to use those values unless overridden by an exact bean-type specification somewhere else in the string, as follows:

```
*=30,100
```

To specify that a default value be used, omit either *min* or *max* but retain the comma (,) between the two values, as follows (split for publication):

```
SMApp#PerfModule#TunerBean=54,
   :SMApp#SMModule#TypeBean=100,200
```

You can specify the bean types in any order within the string.

EJB cache settings

Use this page to configure and manage the cache for a specific EJB container. To determine the cache absolute limit, multiply the number of enterprise beans active in any given transaction by the total number of concurrent transactions expected. Then, add the number of active session bean instances.

To view this administrative console page, click **Servers > Application Servers >** *serverName* **> EJB Container > EJB Cache Settings**.

Cleanup interval

Specifies the interval at which the container attempts to remove unused items from the cache in order to reduce the total number of items to the value of the cache size.

The cache manager tries to maintain some unallocated entries that can be allocated quickly as needed. A background thread attempts to free some entries while maintaining some unallocated entries. If the thread runs while the application server is idle, when the application server needs to allocate new cache entries, it does not pay the performance cost of removing entries from the cache. In general, increase this parameter as the cache size increases.

Data type	Integer
Units	Milliseconds
Range	0 to 2 147 483 674
Default	3000

Cache size

Specifies the number of buckets in the active instance list within the EJB container. A bucket can contain more than one active enterprise bean instance, but performance is maximized if each bucket in the table has a minimum number of instances assigned to it. When the number of active instances within the container exceeds the number of buckets, that is, the cache size, the container periodically attempts to reduce the number of active instances in the table by passivating some of the active instances. For the best balance of performance and memory, set this value to the maximum number of active instances expected during a typical workload.

Data type	Integer
Units	Buckets in the hash table
Range	Greater than 0. The container selects the next largest prime number equal to or greater than the specified value.
Default	2053

Container interoperability

Container interoperability describes the ability of WebSphere Application Server clients and servers at different versions to successfully negotiate differences in native Enterprise JavaBeans (EJB) Version 1.1 finder methods support and Java 2 Platform, Enterprise Edition (J2EE) Version 1.3 compliance.

At one time, there were significant interoperability problems among WebSphere Application Server, versions 4.0.x and 3.5.x distributed, and Version 4.0.x for zSeries. The introduction of interoperable versions of some class types solved these problems for distributed versions 3.5.6, 4.0.3, and 5 as well as for zSeries Version 4.0.x.

Older 4.0.x and 3.5.x client and application server versions do not support the interoperability classes, which makes them uninteroperable with versions that use the classes. The system property *com.ibm.websphere.container.portable* remedies this situation by enabling newer versions of the application server to turn off the interoperability classes. This lets a more recent application server return class types that are interoperable with an older client.

Depending on the value of com.ibm.websphere.container.portable, application servers at versions 5, 4.0.3 and later, and 3.5.6 and later, return different classes for the following:

- Enumerations and collections returned by EJB 1.1 finder methods
- EJBMetaData
- Handles to:
  - Entity beans
  - Session beans
  - Home interfaces

If the property is set to `false`, application servers return the old class types, to enable interoperability with versions 3.5.5 and earlier, and 4.0.2 and earlier. If the property is set to `true`, application servers return the new classes.

Instructions for setting the com.ibm.websphere.container.portable property are in the release notes for versions 3.5.6 and later, and 4.0.3 and later. The following

tables show interoperability characteristics for various version combinations of application servers and clients as well as default property values for each combination.

Interoperability of Version 3.5.x client with Version 5 application server

Clients at Version 3.5.5 and earlier are not interoperable with Version 5 servers when using:

- EJBMetaData
- Enumerations returned by EJB 1.x finder methods
- Handles to entity beans

  If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 3.5.5 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 application server uses the updated handles but returns the enumerations and collections that were used in the earlier clients.

To interoperate with Version 5 application servers, you must upgrade all Version 3.5.x clients to Version 3.5.6 or later.

Interoperability of Version 5 client with Version 3.5.x application server

Client at Version 5, using this function	Application server at Version 3.5.6, property true	Application server at Version 3.5.6, property false (default)	Application server at Version 3.5.5 and earlier
EJBMetaData	Does not work across domains	Works	Does not work
Handle to session bean	Works	Works	Does not work
Handle to entity bean	Does not work across domains	Does not work across domains	Does not work across domains
Enumeration returned by EJB 1.x finder method	Works	Works	Works

Interoperability of Version 4.0.x client with Version 5 application server

Ideally, all 4.0.x clients that use Version 5 application servers should be at Version 4.0.3 or later.

Version 5 application servers return the interoperability class types by default (`true`). This can cause interoperability problems for distributed clients at versions 4.0.1 or 4.0.2. In particular, problems can occur with collections and enumerations returned by EJB 1.1 finder methods.

Although it is strongly discouraged, you can set com.ibm.websphere.container.portable to `false` on a Version 5 application server. This causes the application server to return the old class types, providing interoperability with clients at Version 4.0.2 and earlier. This is discouraged because:

- The Version 5 application server instance would become non-J2EE 1.3 compliant with regard to handles, home interface handles, and EJBMetaData.

- EJB 1.x finder methods return collection and enumeration objects that do not originate from ejbportable.jar.
- Interoperability restrictions still exist with the property set to `false`.
- Version 5 client handles to entity beans and home interfaces do not work across domains for the server you set to `false`.

  If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 4.0.2 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 application server uses the updated handles but returns the enumerations and collections that were used in the earlier clients.

Interoperability of client at Version 4.0.2 and earlier with Version 5 application server

Client at Version 4.0.2 and earlier, using this function	Application server at Version 5, property true (default)	Application server at Version 5, property false
EJBMetaData	Does not work	Works for 4.0.2 client
Handle to session bean	Does not work	Works
Handle to entity bean	Does not work	Does not work across cells
Enumeration returned by EJB 1.x finder method	Does not work	Works
Collection returned by EJB 1.x finder method	Does not work	Works
Handle to home interface	Does not work	Does not work across cells

If you would like to use updated Handle classes in EJB 2.x-compliant beans but have one of the older clients (versions 3.5.5 and earlier, and 4.0.2 and earlier) installed, set the system property com.ibm.websphere.container.portable.finder to `false`. With this setting in place, the Version 5 server uses the new Handle classes but returns the older enumeration and collection classes.

Interoperability of client at Version 4.0.3 and later with Version 5 application server

Clients at Version 4.0.3 and later work well with Version 5 application servers. However, if you set the com.ibm.websphere.container.portable to `false`, client handles to entity beans and home interfaces do not work across domains for the server you set to `false`.

Client at Version 4.0.3 and later, using this function	Application server at Version 5, property true (default)	Application server at Version 5, property false
EJBMetaData	Works	Works
Handle to session bean	Works	Works
Handle to entity bean	Works	Does not work across cells
Enumeration returned by EJB 1.x finder method	Works	Works
Collection returned by EJB 1.x finder method	Works	Works
Handle to home interface	Works	Does not work across cells

Interoperability of Version 5 client with Version 4.0.x application server

Clients at Version 5 work well with Version 4.0.3 application servers if you set com.ibm.websphere.container.portable to `true`. Client handles to entity beans and home interfaces do not work across domains for any Version 4.0.3 server with com.ibm.websphere.container.portable at the default value, `false`. Version 5 client handles to application servers at Version 4.0.2 and earlier also have restrictions.

Client at Version 5, using this function	Application server at Version 4.0.3, property true	Application server at Version 4.0.3, property false (default)	Application server at Version 4.0.2 or earlier
EJBMetaData	Works	Works	Works for 4.0.2 server only
Handle to session bean	Works	Works	Works
Handle to entity bean	Works	Does not work across domains	Does not work across domains
Enumeration returned by EJB 1.x finder method	Works	Works	Works
Collection returned by EJB 1.x finder method	Works	Works	Works
Handle to home interface	Works	Does not work across domains	Does not work across domains

Interoperability of zSeries Version 4.0.x client with Version 5 application server

The only valid configuration for container interoperability with zSeries Version 4.0.x clients is the default configuration for the Version 5 application server.

Interoperability of Version 5 client with zSeries Version 4.0.x application server

Version 5 clients should work with a zSeries Version 4.0.x application server with the correct interoperability fixes described in the zSeries documentation. The interoperability characteristics should be the same as for a Version 4.0.3 distributed application server with the property set to `true`.

Client at Version 5, using this function	zSeries application server at Version 4.0.x
EJBMetaData	Works
Handle to session bean	Works
Handle to entity bean	Works
Enumeration returned by EJB 1.x finder method	Works
Collection returned by EJB 1.x finder method	Works
Handle to home interface	Works

Deploying EJB modules

Assemble one or more EJB modules, assemble one or more Web modules, and assemble them into a J2EE application.

1. Prepare the deployment environment.
2. Deploy the application.
3. 5.0.1 5.0.2 Update the configuration for each EJB module as needed for the deployment environment.
4. For information about the EJB deployment tool, see the EJB deployment tool.

The next step is to test and debug the module.

EJB module collection

Use this page to manage the EJB modules deployed in a specific application.

To view this administrative console page, click **Applications > Enterprise Applications >** *applicationName* **> EJB modules**. Click the check boxes to select one or more of the EJB modules in your collection.

URI

When resolved relative to the application URL, this specifies the location of the module's archive contents on a file system. The URI matches the <ejb> or <web> tag in the <module> tag of the application deployment descriptor.

EJB module settings

Use this page to configure and manage a specific deployed EJB module.

Note: You cannot start or stop an individual EJB module for modification. You must start or stop the appropriate application entirely.

To view this administrative console page, click **Applications > Enterprise Applications >** *applicationName* **> EJB modules >** *moduleName*.

URI

When resolved relative to the application URL, this specifies the location of the module archive contents on a file system. The URI must match the URI of a ModuleRef URI in the deployment descriptor of the deployed application (EAR).

Alternate DD

Specifies a deployment descriptor to be used at run time instead of the one installed in the module.

Starting weight

Specifies the order in which modules are started when the server starts. The module with the lowest starting weight is started first.

Data type	Integer
Default	5000
Range	Greater than 0

Troubleshooting tips for EJBDEPLOY relationships

Problems may exist when EJBDeploy creates a data relationship in DB2 for z/OS Version 7.x. EJBDeploy creates a table with a composite of the two primary keys of

the EJBs that are related to each other. If the composite keys are larger than 254 characters, DB2 for z/OS V7.x will not accept this relationship and the user will be conforonted with errors such as:

```
DSNT408I SQLCODE = -613, ERROR:  THE PRIMARY KEY OR A UNIQUE CONSTRAINT
IS TOO LONG OR HAS TOO MANY COLUMNS
DSNT418I SQLSTATE   = 54008 SQLSTATE RETURN CODE
```

This problem can be seen when the primary keys that are created for the two related beans have primary keys that are strings. This results in the composite being made up of 2 varchar(250) primary keys for a total of 500, which is greater than 254 maximum in DB2 for z/OS version 7.x.

Things to consider when utilizing top-down mappings to ensure you do not experience this problem:
- Top-down mappings are a guideline and must be reviewed with the DBA.
- Schemas created ’top-down’ by EJBDeploy are designed only for testing, and as a guideline for the actual schema required. The use of the ’meet-in-the-middle’ mapping does not present this problem.
- The composite key constraint problem is not experienced when using DB2 V8, which has 2K max key lengths.

Enterprise beans: Resources for learning

Use the following links to find relevant supplemental information about enterprise beans. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- Planning, business scenarios, and IT architecture
- Programming model and decisions
- Programming instructions and examples
- Programming specifications

Planning, business scenarios, and IT architecture
- Mastering Enterprise JavaBeans

  A comprehensive treatment of Enterprise JavaBeans (EJB) programming in nonprintable form (PDF). One must be registered to download the PDF, but registration is free. Information about purchasing a hardcopy is available on the Web site.
- *Enterprise JavaBeans* by Richard Monson-Haefel (O’Reilly and Associates, Inc.: Third Edition, 2001)

Programming model and decisions
- Read all about EJB 2.0

  A comprehensive overview of the specification.
- The J2EE Tutorial

  This set of articles by Sun Microsystems covers several EJB-related topics, including the basic programming models, persistence, and EJB Query Language.

Programming instructions and examples

- Rules and Patterns for Session Facades

  EJB programming practice: Fronting entity beans with a session-bean facade.
- WebSphere Application Server Development Best Practices for Performance and Scalability

  Programming practice for enterprise beans and other types of J2EE components.
- Optimistic Locking in IBM WebSphere Application Server 4.0.2

  Examples of the effect of optimistic concurrency on application behavior. Although the paper is based on a previous version of this product, the data access issues discussed in it are current.

  This paper does not seem to be available directly by URL. To view this paper, visit the specified URL and search on "`optimistic locking`"

Programming specifications

- What's new in the Enterprise JavaBeans 2.0 Specification?

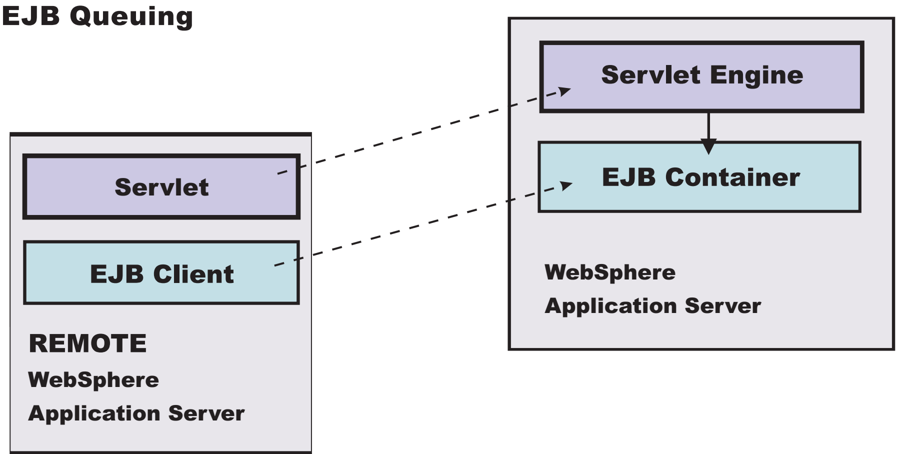
  You can also download the specification itself from this URL.
- JavaTM 2 Platform: Compatibility with Previous Releases

  This Sun Microsystems article includes both source and binary compatibility issues.

EJB method Invocation Queuing

Method invocations to enterprise beans are only queued for remote clients, making the method call. An example of a remote client is an enterprise Java bean (EJB) client running in a separate Java virtual machine (JVM) (another address space) from the enterprise bean. In contrast, no queuing occurs if the EJB client, either a servlet or another enterprise bean, is installed in the same JVM on which the EJB method runs and on the same thread of execution as the EJB client.

Remote enterprise beans communicate by using the Remote Method Invocation over an Internet Inter-Orb Protocol (RMI-IIOP). Method invocations initiated over RMI-IIOP are processed by a server-side object request broker (ORB). The thread pool acts as a queue for incoming requests. However, if a remote method request is issued and there are no more available threads in the thread pool, a new thread is created. After the method request completes the thread is destroyed. Therefore, when the ORB is used to process remote method requests, the EJB container is an open queue, due to the use of unbounded threads. The following illustration depicts the two queuing options of enterprise beans.



Chapter 5. Using message-driven beans in applications

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface.

Message-driven beans (a type of enterprise bean defined in the EJB 2.0 specification) extend the base JMS support and the Enterprise JavaBean component model to provide automatic asynchronous messaging. When a message arrives on a destination, a listener passes the message to a new instance of a user-developed message-driven bean for processing.

You can use WebSphere Studio Application Developer to develop applications that use message-driven beans. You can use the WebSphere Application Server runtime tools, like the administrative console, to deploy and administer applications that use message-driven beans.

For more information about implementing WebSphere enterprise applications that use message-drive beans, see the following topics:
- An overview of message-driven beans
- Designing an enterprise application to use a message-driven bean
- Developing an enterprise application to use a message-driven bean
- Deploying an enterprise application to use a message-driven bean
- Configuring message listener resources for message-driven beans
- Troubleshooting problems with message-driven beans

Message-driven beans - an overview

WebSphere Application Server supports automatic asynchronous messaging with message-driven beans (a type of enterprise bean defined in the EJB 2.0 specification). Messaging with message-driven beans is shown in the figure "Message-driven beans - an overview."

The support for message-driven beans is based on the message listener service, which comprises a listener manager that controls and monitors one or more listeners. Each listener monitors a JMS destination for incoming messages. When a message arrives on the destination, the listener passes the message to a new instance of a user-developed message-driven bean (an enterprise bean) for processing. The listener then looks for the next message without waiting for the bean to return.

Messages arriving at a destination being processed by a listener have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see EJB component security.

You are recommended to develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean, to provide clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.

© Copyright IBM Corp. 2003

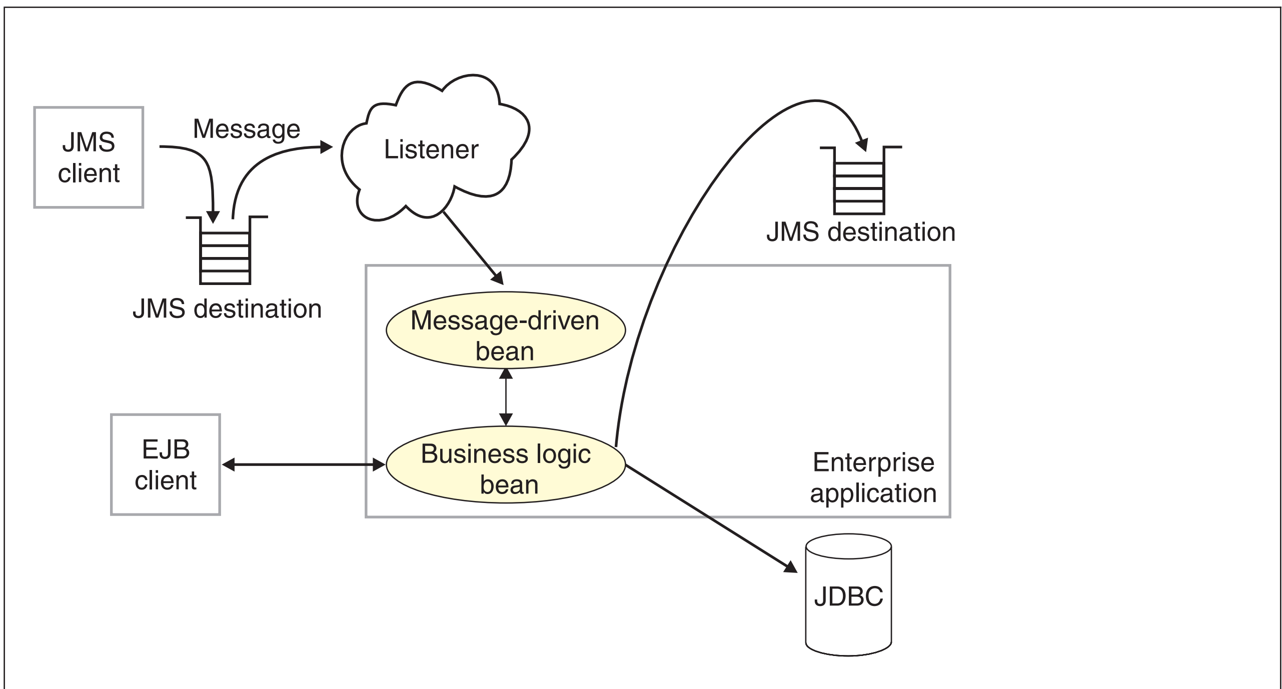


Figure 1. Message-driven beans and the message listener service. This figure shows an incoming message being passed by a JMS listener to a message-driven bean, which passes the message on to a business logic bean for business processing. This messaging is controlled by the listener manager. For more information, see the text that accompanies this figure.

Message-driven beans - components

The WebSphere Application Server support for message-driven beans is based on JMS message listeners and the message listener service, and builds on the base support for JMS. The main components of WebSphere Application Server support for message-driven beans are shown in the following figure and described after the figure:

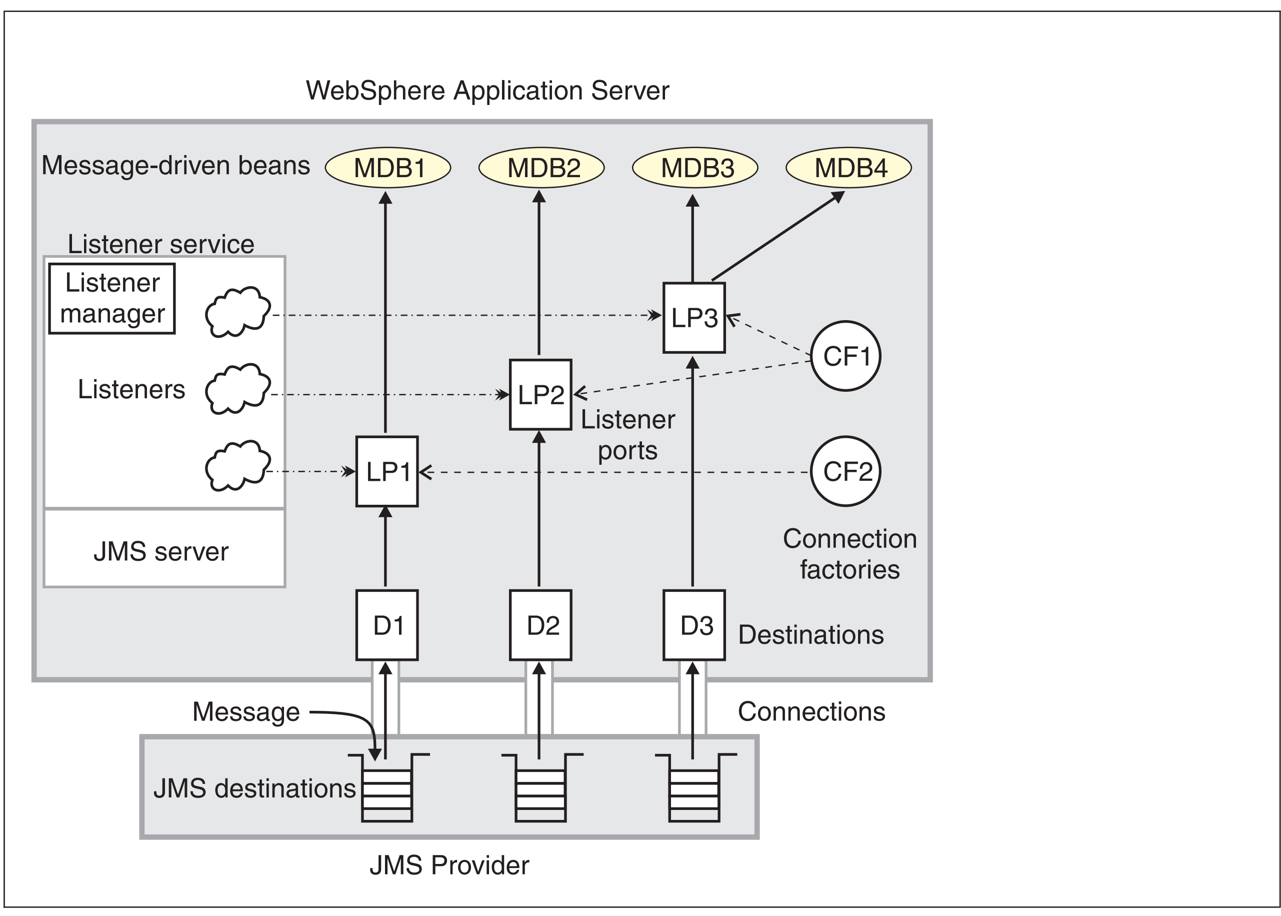


Figure 2. The main components for message-driven beans. This figure shows the main components of WebSphere support for message-driven beans, from JMS provider through a connection to a destination, listener port, then deployed message-driven bean that processes the message retrieved from the destination. Each listener port defines the association between a connection factory, destination, and a deployed message-driven bean. The other main components are the message listener service, which comprises a listener for each listener port, all controlled by the same listener manager. For more information, see the text that accompanies this figure.

The message listener service is an extension to the JMS functions of the JMS provider and provides a listener manager, which controls and monitors one or more JMS listeners.

Each listener monitors either a JMS queue destination (for point-to-point messaging) or a JMS topic destination (for publish/subscribe messaging).

A connection factory is used to create connections with the JMS provider for a specific JMS queue or topic destination. Each connection factory encapsulates the configuration parameters needed to create a connection to a JMS destination.

A listener port defines the association between a connection factory, a destination, and a deployed message-driven bean. Listener ports are used to simplify the administration of the associations between these resources.

When a deployed message-driven bean is installed, it is associated with a listener port and the listener for a destination. When a message arrives on the destination, the listener passes the message to a new instance of a message-driven bean for processing.

When an application server is started, it initializes the listener manager based on the configuration data. The listener manager creates a dynamic session thread pool

for use by listeners, creates and starts listeners, and during server termination controls the cleanup of listener message service resources. Each listener completes several steps for the JMS destination that it is to monitor, including:
- Creating a JMS server session pool, and allocating JMS server sessions and session threads for incoming messages.
- Interfacing with JMS ASF to create JMS connection consumers to listen for incoming messages.
- If specified, starting a transaction and requesting that it is committed (or rolled back) when the EJB method has completed.
- Processing incoming messages by invoking the onMessage() method of the specified enterprise bean.

Message-driven beans - transaction support

Message-driven beans can handle messages read from JMS destinations within the scope of a transaction. If transaction handling is specified for a JMS destination, the JMS listener starts a global transaction *before* it reads any incoming message from that destination. When the message-driven bean processing has finished, the JMS listener commits or rolls back the transaction (using JTA transaction control).

Note:
- All messages retrieved from a specific destination have the same transactional behavior.

If messages are queued to be sent within a global transaction they are sent when the transaction is committed. If the processing of a message causes the transaction to be rolled back, then the message that caused the bean instance to be invoked is left on the JMS destination.

You can configure the **Maximum retries** property of the listener port to define the maximum number of times the listener attempts to read a message from a destination. When the Max retries limit is reached, the listener for that destination is stopped. When you have resolved the problem, you must then restart the listener.

Designing an enterprise application to use message-driven beans

This topic describes things to consider when designing an enterprise application to use message-driven beans.

The considerations in this topic are based on a generic enterprise application that uses one message-driven bean to retrieve messages from a JMS queue destination and passes the messages on to another enterprise bean that implements the business logic.

To design an enterprise application to use message-driven beans, complete the following steps:
1. Identify the JMS resources that the application is to use. This helps to identify the properties of resources that need to be used within the application and configured as application deployment descriptors or within WebSphere Application Server.

JMS resource type	Properties
Queue connection factory	**Name:** SamplePtoPQueueConnectionFactory **JNDI Name:** Sample/JMS/QCF

JMS resource type	Properties
Queue destination	**Name:** Q1 **JNDI Name:** Sample/JMS/Q1
Listener port (for the destination)	**Name:** SamplePtoPListenerPort **Connection Factory JNDI Name:** Sample/JMS/QCF **Destination JNDI Name:** Sample/JMS/Q1 **Maximum Sessions:** 5 **Maximum Retries:** 10 **Maximum Messages:** 1
Message-driven bean (deployment properties)	**Name:** JMSppSampleMDBBean **Transaction type:** Container **Destination type:** Queue **Listener port name:** SamplePtoPListenerPort
Business logic bean	**Name:** MyLogicBean

Ensure that you use consistent values where needed; for example, the JNDI names for the connection factory and destination must be the same for both those resources and the equivalent properties of the listener port.

2. Separation of business logic. You are recommended to develop a message-driven bean to delegate the business processing of incoming messages to another enterprise bean. This provides clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client.
3. Security considerations.

   Messages arriving at a destination being processed by a listener have no client credentials associated with them; the messages are anonymous. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component. For more information about EJB security, see EJB component security.
4. General JMS considerations For Publish/Subscribe messaging, choose the JMS server port to be used depending on your needs for transactions or performance:

   Queued port
   : The TCP/IP port number of the listener port used for all point-to-point and Publish/Subscribe support.

   Direct port
   : The TCP/IP port number of the listener port used for direct TCP/IP connection (non-transactional, non-persistent, and non-durable subscriptions only) for Publish/Subscribe support.

     Note: Message-driven beans cannot use the direct listener port for Publish/Subscribe support. Therefore, any topic connection factory configured with **Port**set to `Direct` cannot be used with message-driven beans.

   A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

Developing an enterprise application to use message-driven beans

Use this task to develop an enterprise application to use a message-driven bean. The message-driven bean is invoked by a JMS listener when a message arrives on the input queue that the listener is monitoring.

You are recommended to develop the message-driven bean to delegate the business processing of incoming messages to another enterprise bean, to provide clear separation of message handling and business processing. This also enables the business processing to be invoked by either the arrival of incoming messages or, for example, from a WebSphere J2EE client. Responses can be handled by another enterprise bean acting as a sender bean, or handled in the message-driven bean.

You develop an enterprise application to use a message-driven bean like any other enterprise bean, except that a message-driven bean does not have a home interface or a remote interface.

This topic describes how to develop a completely new message-driven bean class. If you have a WAS 4.0 enterprise application that uses the JMS listener, you can migrate that application to use message-driven beans.

For more information about writing the message-driven bean class, see *Creating a message-driven bean* in the WebSphere Studio help bookshelf.

To develop an enterprise application to use a message-driven bean, complete the following steps:

1. Creating the Enterprise Application project, as described in the WebSphere Studio article .
2. Creating the message-driven bean class.

   You can use the New Enterprise Bean wizard of WebSphere Studio Application Developer to create an enterprise bean with a bean type of Message-driven bean. The wizard creates appropriate methods for the type of bean.

   By convention, the message bean class is named *name*Bean, where *name* is the name you assign to the message bean; for example:

   ```
   public class MyJMSppMDBBean implements MessageDrivenBean, MessageListener
   ```

   The message-driven bean class must define and implement the following methods:
   - onMessage(message), which must meet the following requirements:
     - The method must have a single argument of type `javax.jms.Message`.
     - The throws clause must *not* define any application exceptions.
     - If the message-driven bean is configured to use bean-managed transactions, it must call the javax.transaction.UserTransaction interface to scope the transactions. Because these calls occur inside the onMessage() method, the transaction scope does not include the initial message receipt. This means the application server is given one attempt to process the message.

     To handle the message within the onMessage() method (for example, to pass the message on to another enterprise bean), you use standard JMS. (This is known as bean-managed messaging.)
   - ejbCreate()

You must define and implement an ejbCreate method for each way in which you want a new instance of an enterprise bean to be created.

- ejbRemove().

  This method is invoked by the container when a client invokes the remove method inherited by the enterprise bean's home interface from the javax.ejb.EJBHome interface. This method must contain any code that you want to execute before an enterprise bean instance is removed from the container (and the associated data is removed from the data source).

For example, the following code extract shows how to access the text and the JMS MessageID, from a JMS message of type TextMessage:
The result of this step is a message-driven bean that can be assembled into an

```
public void onMessage(javax.jms.Message msg)
{
        String text      = null;
        String messageID = null;

        try
        {
                text = ((TextMessage)msg).getText();

                System.out.println("senderBean.onMessage(), msg text2: "+text);

                //
                // store the message id to use as the Correlator value
                //
                messageID = msg.getJMSMessageID();

                // Call a private method to put the message onto another queue
                putMessage(messageID, text);
        }
        catch  (Exception err)
        {
                err.printStackTrace();
        }
        return;
}
```

Figure 3. Code example: The onMessage() method of a message bean. This figure shows a code extract for a basic onMessage() method of a sample message-driven bean. The method unpacks the incoming text message to extract the text and message identifier and calls a private putMessage method (defined within the same message bean class) to put the message onto another queue.

.EAR file for deployment.

3. Assembling and packaging the application for deployment.

   You can use WebSphere Studio to assemble and package the application for deployment.

The result of this task is an .EAR file, containing an application message-driven bean, that can be deployed in WebSphere Application Server.

After you have developed an enterprise application to use message-driven beans, configure and deploy the application; for example, define the listener ports for the message-driven beans and, optionally, change the deployment descriptor attributes for the application. For more information about configuring and deploying an application that uses message-driven beans, see Deploying an enterprise application to use message-driven beans

Deploying an enterprise application to use message-driven beans

Use this task to deploy an enterprise application to use message-driven beans.

This task description assumes that you have an .EAR file, which contains an application enterprise bean with code for message-driven beans, that can be deployed in WebSphere Application Server.

To deploy an enterprise application to use message-driven beans, complete the following steps:

1. Use the WebSphere administrative console to define the listener ports for the application, as described in Adding a new listener port.
2. 5.0.2 For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions, as described in Configuring deployment attributes using the Assembly Toolkit. Alternatively, you can use the Application Assembly Tool as described inConfiguring deployment attributes using the Application Assembly Tool.
3. 5.0.1 For each message-driven bean in the application, configure the deployment attributes to match the listener port definitions, as described in Configuring deployment attributes using the Application Assembly Tool.
4. Use the WebSphere administrative console to install the application.

   This stage is a standard WebSphere Application Server task, as described in Installing a new application.

   When you install the application, you are prompted to specify the name of the listener port that the application is to use for late responses. Select the listener port, then click **OK**.

Configuring deployment attributes using the Assembly Toolkit

Use this task to configure the message-driven beans deployment attributes for an enterprise bean, to override the deployment attributes defined within the application EAR file.

You can configure the deployment attributes of an application by using the Deployment Descriptor Editor of WebSphere Studio Application Developer or the Assembly Toolkit.

This topic describes the use of the Assembly Toolkit to configure the deployment attributes of an application. This task description assumes that you have an EAR file, which contains an application enterprise bean developed as a message-driven bean, that can be deployed in WebSphere Application Server. For more details about using the Assembly Toolkit, see Assembling applications with the Assembly Toolkit.

To configure the message-driven beans deployment attributes for an enterprise bean, use the Assembly Toolkit to configure the deployment attributes of the application to match the listener port definitions:

1. Start the Assembly Toolkit.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the import wizard to import the EAR file into the Assembly Toolkit. To start the import wizard:
   a. Click **File-> Import-> EAR file**
   b. Click **Next**, then select the EAR file.

c. Click **Finish**

3. In the J2EE Hierarchy view, right-click the EJB module for the message-driven bean , then click **Open With > Deployment Descriptor Editor**. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Specify general deployment properties.
   a. In the property pane, select the Beans tab.
   b. Specify the following properties:

      Transaction type
      Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message bean.

      Bean The message bean manages its own transactions

      Container
      The container manages transactions on behalf of the bean
5. Specify advanced deployment properties.
   a. Specify the following properties:

      Message selector
      The JMS message selector to be used to determine which messages the message bean receives; for example:

      ```
      JMSType='car' AND color='blue' AND weight>2500
      ```

      The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

      Acknowledge mode
      How the session acknowledges any messages it receives.

      This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

      Auto Acknowledge
      The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

      Dups OK Acknowledge
      The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

      As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

      Destination type
      Whether the message bean uses a queue or topic destination.

      Queue
      The message bean uses a queue destination.

      Topic The message bean uses a topic destination.

Subscription durability
Whether a JMS topic subscription is durable or non-durable.

Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

Nondurable
Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

6. Specify bindings deployment properties.
   a. Specify the following property:
      Listener port name
      Type the name of the listener port for this message-driven bean.
7. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.
8. Verify the archive files.
9. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
10. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard. Note that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Important: Use **Run On Server** for unit testing only. Assembly Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites the server configuration file for that server. Do not use on production servers.

    For instructions on remote testing, see the article "Setting Up a Remote WebSphere Application Server in WebSphere Studio V5" at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and managing applications.

Configuring deployment attributes for a message-driven bean

Use this task to configure the message-driven beans deployment attributes for an enterprise bean, to override the deployment attributes defined within the application EAR file.

This task description assumes that you have an EAR file, which contains an application enterprise bean developed as a message-driven bean, that can be deployed in WebSphere Application Server.

Note: After deployment code has been generated for an application, the deployable archive is renamed with the prefix Deployed_ . Any subsequent changes to the archive from within the Application Assembly Tool are applied to the version of the archive that existed prior to code generation. To see changes reflected in your application, you must regenerate deployment code and re-install the deployable archive.

To configure the message-driven beans deployment attributes for an enterprise bean, use the the application assembly tool to configure the deployment attributes of the application to match the listener port definitions:

1. Launch the Application Assembly Tool.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, click **File-> Open** then select the the EAR file.
3. In the navigation pane, select the message-driven bean instance; for example, expand *ejb_module_instance*-**> Message-driven beans** then select the bean instance. A property dialog notebook for the message-driven bean is displayed in the property pane.
4. Specify general deployment properties.
   a. In the property pane, select the General tab.
   b. Specify the following properties:
      Transaction type
         Whether the message bean manages its own transactions or the container manages transactions on behalf of the bean. All messages retrieved from a specific destination have the same transactional behavior. To enable the transactional behavior that you want, you must configure the JMS destination with the same transactional behavior as you configure for the message bean.
         Bean The message bean manages its own transactions
         Container
            The container manages transactions on behalf of the bean
5. Specify advanced deployment properties.
   a. In the property pane, select the Advanced tab.
   b. Specify the following properties:
      Message selector
         The JMS message selector to be used to determine which messages the message bean receives; for example:

```
JMSType='car' AND color='blue' AND weight>2500
```

The selector string can refer to fields in the JMS message header and fields in the message properties. Message selectors cannot reference message body values.

Acknowledge mode
How the session acknowledges any messages it receives.

This property applies only to message-driven beans that uses bean-managed transaction demarcation (**Transaction type** is set to `Bean`).

Auto Acknowledge
The session automatically acknowledges a message when it has either successfully returned from a call to receive, or the message listener it has called to process the message successfully returns.

Dups OK Acknowledge
The session lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, so it should be used only by consumers that are tolerant of duplicate messages.

As defined in the EJB specification, clients cannot use using Message.acknowledge() to acknowledge messages. If a value of CLIENT_ACKNOWLEDGE is passed on the create*xxx*Session call, then messages are automatically acknowledged by the application server and Message.acknowledge() is not used.

Destination type
Whether the message bean uses a queue or topic destination.

Queue
The message bean uses a queue destination.

Topic
The message bean uses a topic destination.

Subscription durability
Whether a JMS topic subscription is durable or non-durable.

Durable
A subscriber registers a durable subscription with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the earlier subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

Nondurable
Non-durable subscriptions last for the lifetime of their subscriber object. This means that a client sees the messages published on a topic only while its subscriber is active. If the subscriber is not active, the client is missing messages published on its topic.

A non-durable subscriber can only be used in the same transactional context (for example, a global transaction or an unspecified transaction context) that existed when the subscriber was created. For more information about this context restriction, see The effect of transaction context on non-durable subscribers.

6. Specify bindings deployment properties.
   a. In the property pane, select the Bindings tab.
   b. Specify the following property:

Listener port name
The name of the listener port for this message-driven bean.

7. To apply the changes and close the Application Assembly Tool, click **OK**. Otherwise, to apply the values but keep the property dialog open for additional edits, click **Apply**.
8. To see changes reflected in your application, regenerate deployment code and reinstall the deployable archive.

Configuring message listener resources for message-driven beans

Use the following tasks to configure resources needed by the message listener service to support message-driven beans.

- Configuring the message listener service
- Adding a new listener port
- Configuring a listener port
- Configuring security for message-driven beans

Configuring the message listener service

Use this task to configure the properties of the message listener service for an application server.

To configure the properties of the message listener service for an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. Specify appropriate properties of the message listener service.
5. Optional: Specify any of the following optional properties that you need, as **Custom properties** of the message listener service: NON.ASF.RECEIVE.TIMEOUT, MQJMS.POOLING.TIMEOUT, MQJMS.POOLING.THRESHOLD, MAX.RECOVERY.RETRIES, and RECOVERY.RETRY.INTERVAL.

   For more information about these custom properties, see Custom Properties.
6. Click **OK**.
7. Save your configuration.
8. To have the changed configuration take effect, stop then restart the Application Server.

Message listener service

The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.
This panel displays links to the Additional Properties pages for Listener Ports and Custom Properties for the message listener service.

To view this administrative console page, click **Servers->** ***application_server->*** **Message Listener Service**

Name:

The name by which the message listener service is known for administrative purposes.

Data type	String
Default	MsgLService

Description:

A description of the message listener service, for administrative purposes

Data type	String
Default	Null

Thread pool:

Controls the maximum number of threads the Message Listener Service is allowed to run. Select this link to display the service thread pool properties.

Adjust this parameter when multiple message-driven beans are deployed in the same application server and the sum of their maximum session values exceeds the default value of 10.

Data type	Integer
Units	Not applicable
Default	Minimum: 10, maximum 50
Range	Not applicable
Recommended	Set the minimum to the sum of all message-driven beans maximum session values. Set the maximum to anything equal or greater than the minimum.

Custom Properties:

An optional set of name and value pairs for custom properties of the message listener service.

You can use the Custom properties page to define the following properties for use by the message listener service.
- NON.ASF.RECEIVE.TIMEOUT
- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL

Message listener service custom properties:

Use this panel to view or change an optional set of name and value pairs for custom properties of the message listener service.

To view this administrative console page, click **Servers->** ***application_server*****->** **Message Listener Service-> (In content pane, under Additional Properties) Custom Properties**

You can use the Custom properties page to define the following properties for use by the message listener service.

- NON.ASF.RECEIVE.TIMEOUT
- MQJMS.POOLING.TIMEOUT
- MQJMS.POOLING.THRESHOLD
- MAX.RECOVERY.RETRIES
- RECOVERY.RETRY.INTERVAL

NON.ASF.RECEIVE.TIMEOUT:

The timeout in milliseconds for synchronous message receives performed by message-driven bean listener sessions in the non-ASF mode of operation.

You should set this property to a non-zero value only if you want to enable the non-ASF mode of operation for all message-driven bean listeners on the application server.

The message listener service has two modes of operation, Application Server Facilities (ASF) and non-Application Server Facilities (non-ASF).

- The ASF mode is meant to provide concurrency and transactional support for applications. For publish/subscribe message-drive beans, the ASF mode provides better throughput and concurrency, because in the non-ASF mode the listener is single-threaded.
- The non-ASF mode is mainly for use with generic JMS providers that do not support JMS ASF, which is an optional extension to the JMS specification. The non-ASF mode is also transactional but, because the path length is shorter than the ASF mode, usually provides improved performance.

  Use non-ASF if:
  - Your generic JMS provider does not provide JMS ASF support
  - You are using message-driven beans with WebSphere topic connections with the DIRECT port, because the embedded publish/subscribe broker using that port does not support XA transactions or JMS ASF.
  - Message order is a strict requirement

Data type Integer
Units Milliseconds
Default ASF mode (custom property not created)
Range 0 or greater milliseconds

0 non-ASF mode is disabled

1 or more
The timeout in milliseconds for non-ASF message-driven bean listener synchronous session receives

Recommended	If a transaction timeout occurs, the message must recycle causing extra work. If you want to use the non-ASF mode, set this property to lower than the transaction timeout, but leave spare at least the maximum duration of your message-driven bean's onMessage() method. For example, if your message-driven bean's onMessage() method typically takes a maximum of 10 seconds, and the transaction timeout is set to 120 seconds, you might set the NON.ASF.RECEIVE.TIMEOUT property to no more than 110000 (110000 milliseconds, that is 110 seconds).

MQJMS.POOLING.TIMEOUT:

The number of milliseconds after which a connection in the pool is destroyed if it has not been used.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if it has not been used for five minutes.

Data type	Integer
Units	Milliseconds
Default	5 minutes
Range	

MQJMS.POOLING.THRESHOLD:

The maximum number of unused connections in the pool.

An MQSimpleConnectionManager allocates connections on a most-recently-used basis, and destroys connections on a least-recently-used basis. By default, a connection is destroyed if there are more than ten unused connections in the pool.

Data type	Integer
Units	Number of connections
Default	10
Range	

MAX.RECOVERY.RETRIES:

The maximum number of times that the listener service tries to get a message from a listener port before the associated listener is stopped, in the range 0 through 2147483647.

Data type	Integer
Units	Retry attempts
Default	0 (no retries)
Range	0 (no retries) through 2147483647

RECOVERY.RETRY.INTERVAL:

The time in seconds between retry attempts by the listener service to get a message from a listener port.

Data type	Integer
Units	Seconds
Default	10
Range	1 through 2147483647

Message listener port collection:

The message listener ports configured in the administrative domain

This panel displays a list of the message listener ports configured in the administrative domain. Each listener port is used with a message-driven bean to automatically receive messages from an associated JMS destination. You can use this panel to add new listener ports or to change the properties of existing listener ports. For more information about the property fields for listener ports, see Listener port properties.

To view this administrative console page, click **Servers->** ***application_server*->** **Message Listener Service-> Listener Ports**

Listener port settings:

A listener port is used to simplify administration of the association between a connection factory, destination, and deployed message-driven bean.

Use this panel to view or change the configuration properties of the selected listener port.

To view this administrative console page, click **Servers->** ***application_server*->** **Message Listener Service-> Listener Ports->** ***listener_port***

Name:

The name by which the listener port is known for administrative purposes.

Data type	String
Default	Null

Initial state:

The state that you want the listener port to have when the application server is next restarted

Data type	Enum
Units	Not applicable
Default	Started

Range	**Started** When the application server is next started, the listener port is started automatically. **Stopped** When the application server is next started, the listener port is not started automatically. If message-driven beans are to use this listener port on the application server, the system administrator must start the port manually or select the Started value of this property then restart the application server.

Description:

A description of the listener port, for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

Connection factory JNDI name:

The JNDI name for the JMS connection factory to be used by the listener port; for example, `jms/connFactory1`.

Data type	String
Default	Null

Destination JNDI name:

The JNDI name for the destination to be used by the listener port; for example, `jms/destn1`.

If the extended messaging service is to use this listener port to handle late responses, the value of this property must match the JMS response destination on the output port used by the sender bean.

You cannot use a temporary destination for late responses.

Data type	String
Default	Null

Maximum sessions:

Specifies the maximum number of concurrent sessions that a listener can have with the JMS server to process messages.

Each session corresponds to a separate listener thread and therefore controls the number of concurrently processed messages. Adjust this parameter when the JMS server does not fully use the available capacity of the machine and if you do not need to process messages in a specific message order.

Data type	Integer
Units	Sessions
Default	1
Range	1 through 2147483647
Recommended	• If you want to process messages in a strict message order, set the value to 1, so only one thread is ever processing messages. • If you want to process multiple messages simultaneously (known as "message concurrency"), set this property to a value greater than 1. Keep this value as low as possible to prevent overloading client applications. A good starting point for a 100% JMS workload with short transaction times is 2 to 4 sessions per processor. If longer running transactions exist, you may need more sessions, which should be determined by experimentation.

Maximum retries:

The maximum number of times that the listener tries to deliver a message before the listener is stopped, in the range 0 through 2147483647.

The maximum number of times that the listener tries to deliver a message to a message-driven bean instance before the listener is stopped.

Data type	Integer
Units	Retry attempts
Default	0 (no retries)
Range	0 (no retries) through 2147483647

Maximum messages:

The maximum number of messages that the listener can process in one session with the JMS server.

For WebSphere embedded messaging or WebSphere MQ as the JMS provider, the listener processes all messages in the session as one batch within the same transaction. For a generic JMS provider, the listener processes each message in the session within a separate transaction.

Data type	Integer
Units	Number of messages
Default	1
Range	1 through 2147483647

Recommended	*For WebSphere embedded messaging or WebSphere MQ as the JMS provider*, if you want to process multiple messages in a single transaction, then set this value to more than 1. This enables multiple messages to be batch-processed into a single transaction, and eliminates much of the overhead of transactions on JMS messages. **CAUTION:** • If one message in the batch fails processing with an exception, the entire batch of messages is put back on the queue for processing. • Any resource lock held by any of the interactions for the individual messages are held for the duration of the entire batch. • Depending on the amount of processing that messages need, and if XA transactions are being used, setting a value greater than 1 can cause the transaction to time out. If an XA transaction does time out routinely because processing multiple messages exceeds the transaction timeout, reduce this property to 1 (to limit processing to one message per transaction) or increase your transaction timeout.

Adding a new listener port

Use this task to add a new listener port to the message listener service, so that message-driven beans can be associated with the port to retrieve messages.

To add a new listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, click **New**.
6. Specify appropriate properties for the listener port.
7. Click **OK**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.
9. To have the changed configuration take effect, stop then restart the application server.

If enabled, the listener port is started automatically when a message-driven bean associated with that port is installed.

Configuring a listener port

Use this task to change the properties of an existing listener port, used by message-driven beans associated with the port to retrieve messages.

To configure the properties of a listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, click **Listener Ports**. This displays a list of the listener ports.
5. Click the listener port that you want to modify. This displays the properties of the listener port in the content pane.
6. Specify appropriate properties for the listener port.
7. Click **OK**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.
9. To have the changed configuration take effect, stop then restart the application server.

Deleting a listener port

Use this task to delete a listener port from the message listener service, to prevent message-driven beans associated with the port from retrieving messages.

To delete a listener port, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, select the checkbox for the listener port that you want to delete.
6. Click **Delete**. This action stops the port (needed to allow the port to be deleted) then deletes the port.
7. To save your configuration, click **Save** on the task bar of the Administrative console window.
8. To have the changed configuration take effect, stop then restart the application server.

Configuring security for message-driven beans

Use this task to configure resource security and security permissions for message-driven beans.

Messages arriving at a listener port have no client credentials associated with them. The messages are anonymous.

To call secure enterprise beans from a message-driven bean, the message-driven bean needs to be configured with a RunAs Identity deployment descriptor. Security depends on the role specified by the RunAs Identity for the message-driven bean as an EJB component.

For more information about EJB security, see EJB component security. For more information about configuring security for your application, see Assembling secured applications.

JMS connections used by message-driven beans can benefit from the added security of using J2C container-managed authentication. To enable the use of J2C container authentication aliases and mapping, define a J2C container-managed alias on the JMS connection factory definition that the MDB is using to listen upon (defined by the **Connection factory JNDI name** property of the listener port). If defined, the listener uses the container-managed authentication alias for its JMSConnection security credentials instead of any application-managed alias. To set the container-managed alias, use the administrative console to complete the following steps:

1. To display the listener port settings, click **Servers->** ***application_server*-> Message Listener Service-> Listener Ports->** *listener_port*
2. To get the name of the JMS connection factory, look at the **Connection factory JNDI name** property.
3. Display the JMS connection factory properties. For example, to display the properties of a queue connection factory provided by the embedded WebSphere JMS provider, click **Resources-> WebSphere JMS Provider-> (In content pane, under Additional Properties) WebSphere Queue Connection Factories->** *connection_factory*
4. Set the **Container-managed Authentication Alias** property.
5. Click **OK**

Administering listener ports

Use the following tasks to administer listener ports, which each define the association between a connection factory, a destination, and a message-driven bean.

You can use the WebSphere administrative console to administer listener ports, as described in the following tasks.

- Adding a new listener port

  Use this task to create a new listener port, to specify a new association between a connection factory, a destination, and a message-driven bean. This enables deployed message-driven beans associated with the port to retrieve messages from the destination.
- Configuring a listener port

  Use this task to view or change the configuration properties of a listener port.
- Starting a listener port

  Use this task to start a listener port manually.
- Stopping a listener port

  Use this task to stop a listener port manually.

Note: If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. You do not normally need to start or stop a listener port manually.

Starting a listener port

Use this task to start a listener port on an application server, to enable the listeners for message-driven beans associated with the port to retrieve messages.

A listener is active, that is able to receive messages from a destination, if the deployed message-driven bean, listener port, and message listener service are all started. Although you can start these components in any order, they must all be in a started state before the listener can retrieve messages.

If configured as enabled, a listener port is started automatically when a message-driven bean associated with that port is installed. However, you can start a listener port manually, as described in this topic.

When a listener port is started, the listener manager tries to start the listeners for each message-driven bean associated with the port. If a message-driven bean is stopped, the port is started but the listener is not started, and remains stopped. If you start a message-driven bean, the related listener is started.

To start a listener port on an application server, use the administrative console to complete the following steps:

1. If you want the listener for a deployed message-driven bean to be able to receive messages at the port, check that the message-driven bean has been started.
2. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
3. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.
4. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
5. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
6. Select the checkbox for the listener port that you want to start.
7. Click **Start**.
8. To save your configuration, click **Save** on the task bar of the Administrative console window.

Stopping a listener port

Use this task to stop a listener port on an application server, to prevent the listeners for message-driven beans associated with the port from retrieving messages.

When you stop a listener port as described in this topic, the listener manager stops the listeners for all message-driven beans associated with the port.

To stop a listener port on an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Application Servers** This displays a table of the application servers in the administrative domain.
2. In the content pane, click the name of the application server. This displays the properties of the application server in the content pane.

3. In the Additional Properties table, select **Message Listener Service** This displays the Message Listener Service properties in the content pane.
4. In the content pane, select **Listener Ports**. This displays a list of the listener ports.
5. In the content pane, select the listener port that you want to stop.
6. Click **Stop**.
7. To save your configuration, click **Save** on the task bar of the Administrative console window.
8. To have the changed configuration take effect, stop then restart the application server.

Important files for message-driven beans and extended messaging

The following files in the WAS_HOME/temp directory are important for the operation of the WebSphere Application Server messaging service, so should not be deleted. If you do need to delete the WAS_HOME/temp directory or other files in it, ensure that you preserve the following files.

server_name-**durableSubscriptions.ser**
: You should not delete this file, because the messaging service uses it to keep track of durable subscriptions for message-driven beans. If you uninstall an application that contains a message-driven bean, this file is used to unsubscribe the durable subscription.

server_name-**AsyncMessageRequestLog.ser**
: You should not delete this file, because the messaging service uses it to keep track of late responses that need to be delivered to the late response message handler for the extended messaging provider.

Troubleshooting message-driven beans

Use this overview task to help resolve a problem that you think is related to message-driven beans.

Message-driven beans support uses the standard WebSphere Application Server troubleshooting facilities. If you encounter a problem that you think might be related to the message-driven beans, complete the following stages:

1. Check for messages about message-driven beans in the application server's SystemOut log at *was_home*\logs*server*\SystemOut. Look in the SystemOut log for messages that indicate a problem with JMS resources for message-driven beans, such as listener ports.
2. Check for more messages in the application server's SystemOut log. If the JMS server is running, but you have problems accessing JMS resources, check the SystemOut log file, which should contain more error messages and extra details about the problem.
3. Check the Release Notes for specific problems and workarounds The section *Possible Problems and Suggested Fixes* of the Release Notes, available from the WebSphere Application Server library web site, is updated regularly to contain information about known defects and their workarounds. Check the latest version of the Release Notes for any information about your problem. If the Release Notes does not contain any information about your problem, you can also search the Technotes database on the WebSphere Application Server web site.
4. Check that message listener service has started. The message listener service is an extension to the JMS functions of the JMS provider. It provides a listener

manager that controls and monitors one or more JMS listeners, which each monitor a JMS destination on behalf of a deployed message-driven bean.

5. Check your JMS resource configurations If the WebSphere Messaging functions seem to be running properly (the JMS server is running without problems), check that the JMS resources have been configured correctly. For example, check that the listener ports have been configured correctly and have been started.
6. Check for problems with the WebSphere Messaging functions For more information about troubleshooting WebSphere Messaging, see the related topics.
7. Get a detailed exception dump for messaging. If the information obtained in the preceding steps is still inconclusive, you can enable the application server debug trace for the ″Messaging″ group to provide a detailed exception dump.

Message-driven beans samples

The following examples are provided, as part of the WebSphere Samples Gallery, to illustrate use of the message-driven beans support. When the Samples are installed on your local machine, they are available to try out. Locate them at http://localhost:9080/WSsamples/. (The default port is 9080.) For more information about where to find the Samples Gallery, see Samples Gallery.

- Point-to-point samples:
  - ″Tutorial: Creating JMS message sample″

    This tutorial is designed to help you develop and deploy a JMS message sample application that tests the WebSphere Application Server message-driven beans support in a point-to-point scenario. This sample illustrates how to develop and deploy an application that comprises the following components:
    - A Java/JMS program that writes a message to a queue.
    - A message-driven bean that is invoked by a JMS listener when a message arrives on a defined queue.

    For more information about this sample, see the samples article ″Tutorial: Creating JMS message sample″ that is installed with the Samples option.
  - ″Sample: Message Listener (point-to-point)″

    This sample is designed to demonstrate the use and behavior of message-driven beans for a simple point-to-point scenario. This sample uses the JMS message sample deployed in the sample above.

    For more information about this sample, see the samples article ″Sample: Message Listener (Point-to-Point)″ that is installed with the Samples option.
- Publish/subscribe samples
  - ″Tutorial: Creating JMS message publish/subscribe sample″

    This tutorial is designed to help you develop and deploy a JMS message sample application that tests the WebSphere Application Server message-driven beans support in a publish/subscribe scenario. This sample illustrates how to develop and deploy an application that comprises the following components:
    - A client program that starts the message sequence by publishing a message to a selected topic.
    - A message-driven bean that is invoked by a JMS listener when the broker passes a message to the listener from a topic to which it has subscribed.

    For more information about this sample, see the samples article ″Tutorial: Creating JMS message publish/subscribe sample″ that is installed with the Samples option.

– ″Sample: Message Listener (publish/subscribe)″

This sample is designed to demonstrate the use and behavior of message-driven beans for a simple publish/subscribe scenario. This sample uses the JMS message sample deployed in the publish/subscribe sample above.

For more information about this sample, see the samples article ″Sample: Message Listener (publish/subscribe)″ that is installed with the Samples option.

Chapter 6. Using application clients

An application client module is a JAR (Java ARchive) file containing a client for accessing a Java application.

1. Decide on a type of application client.
2. Develop the application client code.

   Develop J2EE application client code.

   Develop pluggable application client code.
3. Assemble the application client using theAssembly ToolkitorApplication Assembly Tool (AAT).
4. Deploy the application client.

   Deploy the application client on z/OS.

   Deploy the application client on Windows.
5. Run the application client on z/OS or OS/390.

View the Samples gallery for more information about application clients.

Application clients

In a traditional client server environment, the client requests a service and the server fulfills the request. Multiple clients use a single server. Clients can also access several different servers. This model persists for Java clients except now these requests make use of a client run-time environment.

In this model, the client application requires a servlet to communicate with the enterprise bean, and the servlet must reside on the same machine as the WebSphere Application Server.

With WebSphere Application Server Version 5, application clients now consist of the following models:
- ActiveX application client
- Applet client
- J2EE application client
- Pluggable application client
- Thin application client

WebSphere Application Server for z/OS and OS/390 supports only two models:
- J2EE application client
- Pluggable application client

The *ActiveX application client* model, uses the Java Native Interface (JNI) architecture to programmatically access the Java virtual machine (JVM) API. Therefore the JVM code exists in the same process space as the ActiveX application (Visual Basic, VBScript, or Active Server Pages (ASP)) and remains attached to the process until that process terminates.

In the *Applet client* model, a Java applet embeds in a HyperText Markup Language (HTML) document residing on a remote client machine from the WebSphere Application Server. With this type of client, the user accesses an enterprise bean in the WebSphere Application Server through the Java applet in the HTML document.

© Copyright IBM Corp. 2003

The *J2EE application client* is a Java application program that accesses enterprise beans, Java Database Connectivity (JDBC), and Java Message Service message queues. The J2EE application client program runs on client machines. This program follows the same Java programming model as other Java programs; however, the J2EE application client depends on the application client run time to configure its execution environment, and uses the Java Naming and Directory Interface (JNDI) name space to access resources.

The *Pluggable and thin application clients* provide a lightweight Java client programming model. These clients are best suited in situations where a Java client application exists but the application needs enhancements to use enterprise beans, or where the client application requires a thinner, more lightweight environment than the one offered by the J2EE application client. The difference between the thin application client and the pluggable application client is that the thin application client includes a Java virtual machine (JVM) API, and the pluggable application client requires the user to provide this code. The pluggable application client uses the Sun Java Development Kit, and the thin application client uses the IBM Developer Kit For the Java Platform.

The J2EE application client programming model provides the benefits of the J2EE platform for the Java client application. The J2EE application client offers the ability to seamlessly develop, assemble, deploy and launch a client application. The tooling provided with the WebSphere platform supports the seamless integration of these stages to help the developer create a client application from start to finish.

When you develop a client application using and adhering to the J2EE platform, you can put the client application code from one J2EE platform implementation to another. The client application package can require redeployment using each J2EE platform deployment tool, but the code that comprises the client application does not change.

The application client run time supplies a container that provides access to system services for the client application code. The client application code must contain a main method. The application client run time invokes this main method after the environment initializes and runs until the Java virtual machine code terminates.

The J2EE platform allows the application client to use *nicknames* or *short names*, defined within the client application deployment descriptor. These deployment descriptors identify enterprise beans or local resources (JDBC, Java Message Service (JMS), JavaMail and URL APIs) for simplified resolution through JNDI use. This simplified resolution to the enterprise bean reference and local resource reference also eliminates changes to the client application code, when the underlying object or resource either changes or moves to a different server. When these changes occur, the application client can require redeployment.

The application client also provides initialization of the run-time environment for the client application. The deployment descriptor defines this unique initialization for each client application. The application client run time also provides support for security authentication to the enterprise beans and local resources.

The application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access enterprise bean references and to use CORBA services provided by the J2EE platform implementation. Use of the RMI-IIOP protocol and the accessibility of CORBA services assist users in developing a client application that requires access to both enterprise bean references and CORBA object references.

When you combine the J2EE and CORBA environments or programming models in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

View the Samples gallery for more information about application clients. Before you run the basicCalculator Sample, ensure the JMS Server is started.

Application client functions

Use the following table to identify the available functions in the different types of clients.

Note: WebSphere Application Server for z/OS supports only two types of application client:
- J2EE client
- Pluggable client

Available functions	ActiveX client	Applet client	J2EE client	Pluggable client	Thin client
Provides all the benefits of a J2EE platform	Yes	No	Yes	No	No
Portable across all J2EE platforms	No	No	Yes	No	No
Provides the necessary run-time to support communication between client and server	Yes	Yes	Yes	Yes	Yes
Allows the use of nicknames in the deployment descriptor files. **Note:** Although you can edit deployment descriptor files, do not use the administrative console to modify them.	Yes	No	Yes	No	No
Supports use of the RMI-IIOP protocol	Yes	Yes	Yes	Yes	Yes
Browser based application	No	Yes	No	No	No
Enables development of client applications that can access enterprise bean references and CORBA object references	Yes	Yes	Yes	Yes	Yes
Enables the initialization of the client application run-time environment	Yes	No	Yes	No	No
Supports security authentication to enterprise beans	Yes	Limited	Yes	Yes	Yes
Supports security authentication to local resources	Yes	No	Yes	No	No

Requires distribution of application to client machines	Yes	No	Yes	Yes	Yes
Enables access to enterprise beans and other Java classes through Visual Basic, VBScript, and Active Server Pages (ASP) code	Yes	No	No	No	No
Provides a lightweight client suitable for download	No	Yes	No	Yes	Yes
Enables access to Java Naming and Directory Interface (JNDI) for enterprise bean resolution	Yes	Yes	Yes	Yes	Yes
Runs on client machines that use the Sun Java Runtime Environment	No	No	No	Yes	No
Supports CORBA services (using CORBA services can render the application client code nonportable)	No	No	Yes	No	No

J2EE application clients

The J2EE application client programming model provides the benefits of Java TM 2 Platform for WebSphere Application Server Enterprise(J2EE).

The J2EE platform offers the ability to seamlessly develop, assemble, deploy and launch a client application. The tooling provided with the WebSphere platform supports the seamless integration of these stages to help the developer create a client application from start to finish.

When you develop a client application using and adhering to the J2EE platform, you can put the client application code from one J2EE platform implementation to another. The client application package can require redeployment using each J2EE platform deployment tool, but the code that comprises the client application does not change.

The J2EE application client run time supplies a container that provides access to system services for the application client code. The J2EE application client code must contain a main method. The J2EE application client run time invokes this main method after the environment initializes and runs until the Java virtual machine application terminates.

Application clients can use *nicknames* or *short names*, defined within the client application deployment descriptor with the J2EE platform. These deployment descriptors identify enterprise beans or local resources (Java Database Connectivity (JDBC), Java Message Service (JMS), JavaMail and URL APIs) for simplified resolution through JNDI use. This simplified resolution to the enterprise bean reference and local resource reference also eliminates changes to the application client code, when the underlying object or resource either changes or moves to a different server. When these changes occur, the application client can require redeployment. Although you can edit deployment descriptor files, do not use the administrative console to modify them.

The J2EE application client also provides initialization of the run-time environment for the client application. The deployment descriptor defines this unique initialization for each client application. The J2EE application client run time also provides support for security authentication to the enterprise beans and local resources.

The J2EE application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access enterprise bean references and to use CORBA services provided by the J2EE platform implementation. Use of the RMI-IIOP protocol and the accessibility of CORBA services assist users in developing a client application that requires access to both enterprise bean references and CORBA object references.

When you combine the J2EE and the CORBA WebSphere Application Server Enterprise environments or programming models in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

Pluggable application clients

The pluggable application client provides a lightweight, downloadable Java application run time capable of interacting with enterprise beans.

The pluggable application client requires that you have previously installed the Sun Java Runtime Environment (JRE) files. In all other aspects, the pluggable application client, and the thin application client are similar.

Note: *The pluggable client is only available on the Windows platform.*

This client is designed to support those users who want a lightweight Java client application programming environment, without the overhead of the J2EE platform on the client machine. The programming model for this client is heavily influenced by the CORBA programming model, but supports access to enterprise beans.

When accessing enterprise beans from this client, the client application can consider the enterprise beans object references as CORBA object references.

Tooling does not exist on the client; however, tooling does exists on the server. You are responsible for developing the client application, generating the necessary client bindings for the enterprise bean and CORBA objects, and after bundling these pieces together, installing them on the client machine.

The pluggable application client provides the necessary run time to support the communication needs between the client and the server.

The pluggable application client uses the RMI-IIOP protocol. Using this protocol enables the client application to access not only enterprise bean references and CORBA object references, but the protocol also allows the client application to use any supported CORBA services. Using the RMI-IIOP protocol along with the accessibility of CORBA services can assist a user in developing a client application that needs to access both enterprise bean references and CORBA object references.

When you combine the J2EE and CORBA environments in one client application, you must understand the differences between the two programming models to use and manage each appropriately.

The pluggable application client run time provides the necessary support for the client application for object resolution, security, Reliability Availability and Serviceability (RAS), and other services. However, this client does not support a container that provides easy access to these services. For example, no support exists for using *nicknames* for enterprise beans or local resource resolution. When resolving to an enterprise bean (using either Java Naming and Directory Interface (JNDI) or CosNaming) sources, the client application must know the location of the name server and the fully qualified name used when the reference was bound into the name space. When resolving to a local resource, the client application cannot resolve to the resource through a JNDI lookup. Instead the client application must explicitly create the connection to the resource using the appropriate API (JDBC, Java Message Service (JMS), and so on). This client does not perform initialization of any of the services that the client application might require. For example, the client application is responsible for the initialization of the naming service, either through CosNaming or JNDI APIs.

The pluggable application client offers access to most of the available client services in the J2EE application client. However, you cannot access the services in the pluggable client as easily as you can in the J2EE application client. The J2EE client has the advantage of performing a simple Java Naming and Directory Interface (JNDI) name space lookup to access the desired service or resource. The pluggable client must code explicitly for each resource in the client application. For example, looking up an enterprise bean Home requires the following code in a J2EE application client:

```
        java.lang.Object ejbHome =  initialContext.lookup("java:/comp/env/ejb/MyEJBHome"
);
    MyEJBHome = (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome,
MyEJBHome.class);
```

However, you need more explicit code in a Java pluggable application client:

```
        java.lang.Object ejbHome =  initialContext.lookup("the/fully/qualified
/path/to/actual/home/in/namespace/MyEJBHome");
    MyEJBHome = (MyEJBHome)javax.rmi.PortableRemoteObject.narrow(ejbHome,
MyEJBHome.class);
```

In this example, the J2EE application client accesses a logical name from the `java:/comp` name space. The J2EE client run time resolves that name to the physical location and returns the reference to the client application. The pluggable client must know the fully qualified physical location of the enterprise bean Home in the name space. If this location changes, the pluggable client application must also change the value placed on the lookup() statement.

In the J2EE client, the client application is protected from these changes because it uses the logical name. A change can require a redeployment of the EAR file, but the actual client application code remains the same.

The pluggable application client is a traditional Java application that contains a *main* function. The WebSphere pluggable application client provides run time support for accessing remote enterprise beans, and provides the implementation for various services (security, Workload Management (WLM), and others). This client can also access CORBA objects and CORBA based services. When using both environments in one client application, you need to understand the differences between the enterprise bean and CORBA programming models to manage both environments.

For instance, the CORBA programming model requires the CORBA CosNaming name service for object resolution in a name space. The enterprise beans programming model requires the JNDI name service. The client application must initialize and properly manage these two naming services.

Another difference applies to the enterprise bean model. Use the Java Naming and Directory Interface (JNDI) implementation in the enterprise bean model to initialize the Object Request Broker (ORB). The client application is unaware that an ORB is present. The CORBA model, however, requires the client application to explicitly initialize the ORB through the `ORB.init()` static method.

The pluggable application client provides a batch command that you can use to set the CLASSPATH and JAVA_HOME environment variables to enable the pluggable application client run time.

Migration tips for application clients

Tips for migrating thin application client code:
- The Java invocation used to run non-J2EE application clients has changed in Version 5.0. You must specify `-Xbootclasspath/p:%WAS_BOOTCLASSPATH%` on Windows systems . Set the `WAS_BOOTCLASSPATH` environment variable in one of the following:
  - `setupClient.bat` for Windows systems .
  - `setupCmdLine.bat` for Windows systems .

  For more information about using `-Xbootclasspath`, view sample code at the following path after you preform the application client installation:

  ```
  install_root\samples\bin\ActiveXBridgeClients\VB\XJBExamples\modXJBHelpers.bas
  ```

Tips for migrating J2EE application client code:
- If your J2EE application client uses resource references and you have configured those resources using the Application Client Resource Configuration Tool (ACRCT), you must run the **ClientUpgrade** command to migrate the resource configuration information in WebSphere Application Server V5.

Developing J2EE application client code

A *J2EE application client* program operates similarly to a standard J2EE program in that it runs its own ASCII Java Virtual Machine code and is invoked at its main method. This JVM run-time environment is part of the client container, which provides the following services for the application client:
- Security
- Communications protocol support (for RMI/IIOP, HTTP, and so on)
- Naming support

The Java Virtual Machine application client program differs from a standard Java program because it uses the Java Naming and Directory Interface (JNDI) name space to access resources. In a standard Java program, the resource information is coded in the program.

Storing the resource information separately from the client application program makes the client application program portable and more flexible.

1. Writing the client application program. Write the J2EE application client program on any development machine. At this stage, you do not require access to the WebSphere Application Server.

Rules: If you are writing a client application program that will run on z/OS, the following rules apply:

- Client programs may start their own transactions but cannot join in or start transactions in the WebSphere Application Server for z/OS run-time.
- Application client code must contain a main method.
- All input and output files for the application client must be in ASCII, because the client run-time runs in an ASCII JVM.

Using the `javax.naming.InitialContext` class, the client application program uses the lookup operation to access the Java Naming and Directory Interface (JNDI) name space. The `InitialContext` class provides the `lookup` method to locate resources.

The following example illustrates how a client application program uses the `InitialContext` class:

```
import javax.naming.*

public class myAppClient
{
    public static void main(String argv[])
    {
        InitialContext initCtx = new InitialContext();
        Object homeObject = initCtx.lookup("java:comp/env/ejb/BasicCalculator");
        BasicCalculatorHome bcHome = (BasicCalculatorHome)
        javax.rmi.PortableRemoteObject.narrow(homeObject, BasicCalculatorHome.class);
        BasicCalculatorHome bc = bcHome.create();                          ...
    }
}
```

In this example, the program looks up an enterprise bean called `BasicCalculator`. The `BasicCalculator` EJB reference is located in the client JNDI name space at `java:comp/env/ejb/BasicCalculator` . Since the actual enterprise bean runs on the server, the application client run time returns a reference to the `BasicCalculator` home interface.

If the client application program lookup was for a resource reference or an environment entry, then lookup returns an instance of the configured type as defined by the client application deployment descriptor. For example, if the program lookup was a JDBC datasource, the lookup would return an instance of `javax.sql.DataSource`. Although you can edit deployment descriptor files, do not use the administrative console to modify them.

2. Assemble the application client using the Assembly Toolkitor the Application Assembly Tool (AAT).

   The JNDI name space knows what to return on a lookup because of the information assembled by the assembly tool.

   Assemble the J2EE application client on any development machine with the assembly tool installed.

   When you assemble your application client, provide the *application client* run time with the required information to initialize the execution environment for your client application program. Refer to the Application Assembly Tool description for implementation details.

   Keep the following in mind when you configure resources used by your client application program:

   - **5.0.1** **5.0.2** When configuring resource references, resource environment references, and EJB references in the Application Assembly Tool, the `General` tab contains a required `Name` field. This field specifies where the application

client run time binds the reference to the real object in the `java:comp/env` portion of the JNDI name space. The application client run time always binds these references relative to `java:comp/env`. For the programming example above, specify `ejb/BasicCalculator` in the `Name` field on the `General` tab of the Application Assembly Tool, which requires the program to perform a lookup of `java:comp/env/ejb/BasicCalculator`. If the `Name` field is set to `myString`, the resulting lookup is `java:comp/env/myString`.

- **5.0.1** **5.0.2** When configuring Resource references in the Application Assembly Tool, the `Name` field on the `General` tab is used for:
  - **5.0.1** **5.0.2** Binding a reference of that object type into the JNDI name space.
  - **5.0.1** **5.0.2** Retrieving client specific resource configuration information that was configured using the Application Client Resource Configuration Tool(ACRCT) on Windows, or the ACRCT scripting tool on z/OS.
- **5.0.1** **5.0.2** When configuring a resource reference in the Application Assembly Tool, the value in the `Name` field on the `General` tab must match the value in the JNDI `Name` field on the `General` tab for the resource in the Application Client Resource Configuration Tool.
- **5.0.1** **5.0.2** When configuring URL resources using the Application Client Resource Configuration Tool, the URL provider panel enables you to specify a protocol and a class to handle that protocol. If you want to use the default protocols, such as HTTP, you can leave those fields blank.
- **5.0.1** **5.0.2** When configuring resource references using the Application Assembly Tool, the `General` tab contains a field called `Authorization`. You can set this field to either `Container` or `Application`. If you set the field to `Container`, then the application client run time uses authorization information configured in the Application Client Resource Configuration tool for the resource. If the field is set to `Application`, then the application client run time expects the user application to provide authorization information for the resource. The application client run time ignores any authorization information configured with the Application Client Resource Configuration tool for that resource.
- **5.0.1** **5.0.2** When configuring resource environment references using the Application Assembly Tool, you must specify the location of the actual object in the server JNDI namespace using the Binding tab. A resource environment reference maps a logical name (the `Name` field on the general tab) used by the client application to the physical name of an object (the JNDI Name field on the Bindings tab). Not all objects bound into the server JNDI namespace are intended for use by an application client. For example, the WebSphere Application Server client run-time does not support the use of Java 2 Connector (J2C) objects on the client. The object needs to be remotable, and the client-side implementations must be made available on the application client run-time classpath.
- Resource environment references are different than resource references. Resource environment references allow your application client to use a logical name to look-up a resource bound into the server JNDI namespace. A resource reference allows your application to use a logical name to look-up a local J2EE resource. The J2EE specification does not specify a particular implementation of a resource. The following is a table of the supported resource types and identifies the resources to which the WebSphere Application Server provides a client implementation.

Resource Type	Client Configuration Notes	Client implementation provided by WebSphere Application Server
javax.sql.DataSource	Supports specification of any Datasource implementation class	No
java.net.URL	Supports specification of custom protocol handlers	Provided by Java Runtime Environment files
javax.mail.Session	Supports custom protocol configuration	Yes - POP3, SMTP, IMAP
javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory, javax.jms.Queue, javax.jms.Topic	Supports configuration of WebSphere Embedded Messaging, IBM MQ Series and other JMS providers	Yes - WebSphere Embedded Messaging

3. Assemble the Enterprise Archive (EAR) file.

   The application is contained in an enterprise archive or `.ear` file. The `.ear` file is composed of:
   - Enterprise bean, application client, and user-defined modules or `.jar` files
   - Web applications or `.war` files
   - Metadata describing the applications or application `.xml` files

   You must assemble the `.ear` file on the server machine.
4. Distribute the EAR file

   The client machines configured to run this client must have access to the `.ear` file.

   You can either distribute the `.ear` files to the correct client machines, or make them available on a network drive.

   Distributing the `.ear` files is the responsibility of the system and network administrator.
5. Deploy the application client.

   If you plan to deploy the client on z/OS or OS/390, you have two options for running the Application Client Resource Configuration Tool (ACRCT):
   - Run the ACRCT on Windows, or
   - Run the ACRCT scripting tool on z/OS.

   Both of these options produce equivalent output; only the tool interfaces are different. The ACRCT on Windows presents a graphical user interface, whereas the ACRCT for z/OS uses a scripting interface.
6. Use the WebSphere Administrative console to install the application client on z/OS or OS/390.

After completing these steps, launch the application client.

J2EE application client class loading

When you run your J2EE application client, a hierarchy of class loaders is created to load classes used by your application.

The following list describes the hierarchy of class loaders:

- The topmost class loader, the `bootstrap class loader`, contains the JAR files that make up the Java Virtual Machine code, such as `rt.jar`, plus those JAR files defined by the `-Xbootclasspath` parameter on the Java command. The WebSphere Application client run time sets this value to the `WAS_BOOTCLASSPATH` environment variable.
- The *extensions class loader* class loader is a child to the bootstrap class loader. This class loader contains JAR files in the `java/jre/lib/ext` directory or those JAR files defined by the `-Djava.ext.dirs` parameter on the Java command. The WebSphere Application Client run time does not set `-Djava.ext.dirs` parameters, so it uses the JAR files in the `java/jre/lib/ext` directory.
- The *system class loader* class loader contains JAR files and classes that are defined by the `-classpath` parameter on the java command. The Application Client run time sets this parameter to the `WAS_CLASSPATH` environment variable.
- The *WebSphere class loader* class loader loads the WebSphere Application Client run time and any classes placed in the WebSphere Application Client user directories. The directories used by this class loader are defined by the `WAS_EXT_DIRS` environment variable. The `WAS_BOOTCLASSPATH`, `WAS_CLASSPATH`, and the `WAS_EXT_DIRS` environment variables are set in the `installation_root/bin/setupCmdLine` command shell for WebSphere Application Server server installations, or in the `installation_root/bin/setupClient` command shell for client installations.

As the J2EE application client run time initializes, additional class loaders are created as children of the WebSphere class loader. If your client application uses resources such as Java Database Connectivity (JDBC) API, Java Message Service (JMS) API, or Uniform Resource Locator (URL), a different class loader is created to load each of those resources. Finally, the application client run time sets the WebSphere class loader to load classes within the `.ear` file by processing the client JAR manfest repeatedly. The system classpath, defined by the CLASSPATH environment variable is never used and is not part of the hierarchy of class loaders.

To package your client application correctly, you must understand which class loader loads your classes. When Java loads a class, the class loader used to load that class is assigned to it. Any classes subsequently loaded by that class will use that class loader or any of its parents, but it will not use children class loaders.

In some cases the WebSphere Application Client run time can detect when your client application class is loaded by a different class loader from the one created for it by the WebSphere Application Client run time. When this detection occurs, you see the following message:

```
WSCL0205W: The incorrect class loader was used to load [0]
```

This message occurs when your client application class is loaded by one of the parent class loaders in the hierarchy. This situation is typically caused by having the same classes in the `.ear` file and on the hard drive. If one of the parent class loaders locates a class, that class loader loads it before the application client run time class loader. In some cases, your client application will still function correctly. In most cases, however, you receive "class not found" exceptions.

Configuring the classpath fields

When packaging your J2EE client application, you must configure various classpath fields. Ideally, you should package everything required by your application into your `.ear` file. This is the easiest way to distribute your J2EE client application to your clients. However, you should not package such resources as

JDBC APIs, JMS APIs, or URLs. In the case of these resources, use classpath references to access those classes on the hard drive. You might also have other classes installed on your client machines that you do not need to redistribute. In this case, you also want to use classpath references to access the classes on the hard drive, as described below.

Referencing classes within the EAR file

WebSphere J2EE applications do not use the system class path. Use the MANIFEST Classpath entry to refer to other JARs within the `.ear` file. Configure these values using the module Classpath fields in the Application Assembly Tool. For example, if your client application needs to access the path of the enterprise bean JAR, add the deployed enterprise bean module name to your application client Classpath field in the Application Assembly Tool. The format of the Classpath field for each of the different modules (Application Client, enterprise bean, Web) is the same:

- The values must refer to `.jar` and `.class` files that are contained within the `.ear` file.
- The values must be relative to the root of the `.ear` file.
- The values cannot refer to absolute paths in the file systems.
- Multiple values must be separated by spaces, not colons or semi-colons.

Note: This is the Java method for allowing applications to function platform-independent.

Typically, you add modules (`.jar` files) to the root of the `.ear` file. In this case, you only need to specify the name of the module (`.jar` file) in the Classpath field. If you choose to add a module with a path, you need to specify the path relative to the root of the `.ear` file.

For referencing `.class` files, you must specify the directory relative to the root of the `.ear` file. With the Application Assembly Tool you can add individual class files to the `.ear` file. It is recommended that these additional class files are packaged in a `.jar` file. Add this `.jar` file to the module Classpath fields. If you add `.class` files to the root of the `.ear` file, add **./** to the module Classpath fields. Consider the following example directory structure in which the file myapp.ear contains an application client JAR file named `client.jar` and a `mybeans.jar` EJB module. Additional classes reside in class1.jar and utility/class2.zip files. A class named xyz.class is not packaged in a JAR file but is in the root of the EAR file. Specify **./ mybeans.jar utility/class2.zip class1.jar** as the value of the Classpath property. The search order is: `myapp.ear/client.jar` `myapp.ear/xyz.class` `myapp.ear/mybeans.jar` `myapp.ear/utility/class2.zip` `myapp.ear/class1.jar`

Resource classpaths

When you configure resources used by your client application using the Application Client Resource Configuration Tool(ACRCT), or the z/OS ACRCT scripting tool, you can specify classpaths that are required by the resource. For example, if your application is using a JDBC to a DB2 database, add `db2java.zip` to the classpath field of the database provider. These classpath values are platform-specific and require semi-colons or colons to separate multiple values.

Developing pluggable application client code

As you prepare to install the pluggable application client, remember that pluggable clients are only available on Windows systems.

1. Install the pluggable application client from the WebSphere Application Client CD by selecting option **Pluggable Application Client** from the **Custom client installation** panel.
2. Set the Java application pluggable client environment by using the **setupClient** shell, located in:

   ```
   install_root\AppClient\bin\setupClient.bat
   ```
3. Add your specific Java client application JAR files to the CLASSPATH and start your Java client application from this environment, after setting the environment variables.
4. Run the following Java command to invoke your client application:

   ```
   %JAVA_HOME%/bin/java -Xbootclasspath/p:%WAS_BOOTCLASSPATH% -classpath
   <list of your application jars and classes> -Djava.ext.dirs=%WAS_EXT_DIRS%
   -Djava.naming.provider.url=iiop://<your WebSphere server machine name>
   -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
   %SERVER_ROOT% %CLIENTSAS% <fully qualified class name to run>
   ```

   ```
   $JAVA_HOME/bin/java -Xbootclasspath/p:$WAS_BOOTCLASSPATH -classpath
   <list of your application jars and classes> -Djava.ext.dirs=$WAS_EXT_DIRS
   -Djava.naming.provider.url=iiop://<your WebSphere server machine name>
   -Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContextFactory
   $SERVER_ROOT $CLIENTSAS <fully qualified class name to run>
   ```

View the Samples gallery for more information about application clients. Before you run the `basicCalculator` Sample, ensure the JMS Server is started.

Assembling application clients

Assemble a client module to contain application client code. Group enterprise beans, Web components, and resource adapter code in separate modules.

Application client projects contain programs that run on networked client systems. An application client project is deployed as a JAR file.

Use the Assembly Toolkit to assemble an application client module in any of the following ways:
- Import an existing application client JAR file.
- Create a new application client module.

1. Start the Assembly Toolkit.
2. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
4. Migrate application client JAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import your application client JAR files to the Assembly Toolkit.
5. Create a new application client.
6. Verify the contents of the new application client in either of the following ways:
   - In the J2EE Hierarchy view, expand **Application Client Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the application client module in a Navigator view.

After you finish assembling all of your application's modules, you are ready to deploy your application:
- On Windows; or
- On z/OS or OS/390.

Assembling Application Client Modules

If you want to use existing J2EE 1.2 Web modules in your J2EE 1.3 application, migrate them to J2EE 1.3 first.

Note: This task only applies to J2EE application clients.

Assemble a client module to contain application client code. (Group enterprise beans, Web components, and resource adapter code in separate modules).

1. Start the AAT.
2. From the New tab, select **Application Client**. Click **OK**. The navigation tree now displays various sets of properties for configuring the new application client.
3. Use the property dialog shown in the AAT workspace to change the default file name and location.
   a. It is recommended that you change the display name so that it differs from the file name.
   b. If you like, change the temporary location of the application client from the default location, `install_root/bin`.
4. Enter the main class filename and location.
   a. Click **Browse** to locate the class file.
   b. Select the archive containing the class files, and click **Select**.
   c. Select the files you need from the archive, and click **OK**.
5. Define the assembly properties for the application client.
   a. Right-click the assembly property in the navigation pane.
   b. Select **New** in the right-click menu.
   c. Enter values in the property dialog box.
   d. Click **OK**.
6. Add files for the application client.
   a. Right-click **Files** in the navigation pane.
   b. Select **Add Files** from the right-click menu.
   c. Locate the directory where the files are located and click **Select**.
   d. Select the files to add and click **Add**.
   e. In the **Selected files** window, click **OK**.
7. Save the application.

Assemble other new modules of your choice, if needed:
- Assembling EJB modules.
- Assembling resource adapter modules.

Another option is to proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

After you finish assembling all of your application's modules, you are ready to deploy your application:
- on Windows, or
- on z/OS or OS/390.

Application client assembly settings

Use this page to specify assembly properties for J2EE application clients.

File name (Required, String)

Specifies the file name of the application client module, relative to the top level of the Enterprise Archive (EAR) file.
If this is a stand-alone module, the file name is the full path name of the archive.

Alternative DD

Specifies the file name for an alternative deployment descriptor file to use instead of the original deployment descriptor file in the module Java Archive (JAR) file. This file is the post-assembly version of the deployment descriptor file. (Manually edit the original deployment descriptor file to resolve dependencies and security information. Directing the use of the alternative deployment descriptor allows you to keep the original deployment descriptor file intact). The value of the Alternative DD property must be the full path name of the deployment descriptor file relative to the module root directory. By convention, the file is in the `ALT-INF` directory. If this property is not specified, the deployment descriptor file is read directly from the module JAR file.

Classpath

Specifies the full classpath containing the dependent code that is not contained in the application client JAR file.
Specify the values relative to the root of the EAR file and separate the values with spaces. Absolute values that reference files or directories on the hard drive are ignored. To specify classes that are not in JAR files but are in the root of the EAR file, use a period and forward slash (./). Consider the following example directory structure in which the file myapp.ear contains an application client JAR file named `client.jar`. Additional classes reside in `class1.jar` and `class2.zip` files. A class named xyz.class is not packaged in a JAR file but is in the root of the EAR file `myapp.ear/client.jar myapp.ear/class1.jar myapp.ear/class2.zip myapp.ear/xyz.class`. Specify class1.jar class2.zip ./ as the value of the Classpath property. (Name only the directory for `.class` files.)

Display name (Required, String)

Specifies a short name that is intended for display by GUIs.

Small icon

Specifies a JPEG or GIF file containing a small image (16x16 pixels).
The image is used as an icon to represent the application client in a GUI.

Large icon

Specifies a JPEG or GIF file containing a large image (32x32 pixels).
The image is used as an icon to represent the application client in a GUI

Description

Contains text describing the application client.

Main class (Required, String)

Specifies the full path name of the main class for this application client.

Deploying application clients on z/OS

For J2EE application clients that will run on z/OS or OS/390, you may use one of the following options to define resources:
- Run the Application Client Resource Configuration Tool (ACRCT) on Windows.
- Run the ACRCT scripting tool on z/OS.

Both options produce identical output with one possible exception: the sequence in which resource definitions are stored in the Enterprise Archive (EAR) file of the application client. The client container on z/OS uses these resource definitions for resolving and creating an instance of the resources for the application client.

Before you begin: Make sure you have completed the following tasks:
1. Develop the J2EE application client according to guidelines.
2. Assemble the application client.
3. Find out what resources are available on the z/OS system on which you will install the client. These resources include:
   - Enterprise beans
   - JMS message resources
   - JDBC databases
   - Java Mail providers
   - Environment entries (native types)
   - URLs
4. Decide whether you want to provide resource properties for the ACRCT scripting tool on the command line or through an input file. If you do not specify required properties, the ACRCT scripting tool will issue an error message to the MVS console and will end its processing.

   Recommendation: Determine which resource or provider properties are required.

1. Use the WebSphere Administrative console to install the application client on z/OS or OS/390.
2. (Optional) Set up a plain-text input file to provide on the command line when you start the ACRCT scripting tool.

   Rules:
   - Each line in the input file may contain only one key and value pair that defines a property of the resource to be configured.
   - For each resource to be configured, determine which resource or provider properties are required.
   - Follow the syntax rules explained in Application Client Resource Configuration Tool (ACRCT) Scripting tool for z/OS.
   - You may define your own properties for the resource, using the format property.name=value

   Sample: Input file for a data source provider:

   ```
   providertype=DataSourceProvider
   providername=DB2UDBV7
   name="PolicyDatasource"
   description="Datasource for Policy App"
   jndiname=jdbc/PolicyDS
   databasename=POLICYAPP
   user=dbuser
   password=dbpw
   reenterpassword=dbpw
   property.my.resource.property.one=value1
   property.my.resource.property.two=value2
   ```
3. On z/OS or OS/390, start the ACRCT scripting tool by invoking the shell script clientConfig in the UNIX System Services (USS) environment.

   Example:

   ```
   /usr/lpp/WebSphere/V5R0M0/bin/clientConfig.sh
   ```

Rule: You must specify the application client's Enterprise Archive (EAR) file on the command line. You may either specify resource parameters directly on the command line, or specify an input file. The syntax and parameter descriptions appear in Application Client Resource Configuration Tool (ACRCT) Scripting tool for z/OS.

- If the resource parameters are properly specified, the ACRCT scripting tool updates the application client's client-resources.xmi file with appropriate resource definitions.
- If the resource parameters are not properly specified or are missing, the ACRCT scripting tool issues an error message to the MVS console, and ends its processing.

Tip: If you receive an error message in response to the invocation, consider using the help function described in Application Client Resource Configuration Tool (ACRCT) Scripting tool for z/OS.

When the scripting tool successfully completes, the application client's EAR file is updated with the appropriate resource definitions.

When you have finished defining or updating the application client's resources, launch the application client.

Application Client Resource Configuration Scripting tool for z/OS

This section describes the command line syntax for the z/OS scripting version of the Application Client Resource Configuration Tool (ACRCT). The ACRCT scripting tool for z/OS allows you to:

- Define or delete resources for an application client that will run on z/OS.
- List the properties of a resource or provider that is already defined for the application client.

For define and delete actions, the ACRCT scripting tool alters the Enterprise Archive (EAR) file for the application client, as instructed by options you specify. If the ACRCT scripting tool encounters an error at any time during its processing, the tool issues an error message to the MVS console and terminates without changing the original contents of the application client EAR file.

When you use the ACRCT scripting tool, you may specify:

- More than one action (define, delete, or list) to perform for a specific application client.
- More than one EAR file, to define or delete resources for more than one application client at a time.

The command line invocation syntax for the ACRCT scripting tool follows. When you have a choice of one required keyword, those keywords appear within brackets [].

- To define or delete resources:

```
acrct -earfile earfile [-define | -delete] [-provider | -resource]
[-f inputfile | key=value]
```

- To list resources:

```
acrct -list [-provider | -resource] [-f inputfile | -p key=value]
```

- To get help information:

```
acrct -help
```

Parameters

where:

-earfile
Is a required parameter that indicates the input filename of the application client EAR file.

earfilename
Identifies the location and name of the EAR file that contains the application client. This path and filename must directly follow the -earfile parameter.

-define
Instructs the scripting tool to define a provider or resource based on the input properties.

-delete
Instructs the scripting tool to delete a provider or resource based on the input properties.

-list
Instructs the scripting tool to the properties of a particular provider or resource, based on the input properties.

-help
Instructs the scripting tool to list basic examples and guidelines for using quotes around key values.

-provider
Indicates that the object to be defined or deleted, or for which properties are to be listed, is a provider.

-resource
Indicates that the object to be defined or deleted, or for which properties are to be listed, is a resource.

-f Indicates that the input properties for the provider or resource are provided in an input file, rather than specified directly on the command line.

inputfile
Identifies the location and name of the input file that contains the provider or resource properties. This path and filename must directly follow the -f parameter.

key=*value*
Specifies an input property for the provider or resource, in the form of key and value pairs.

Rules:

- You must use lowercase for keys.
- You cannot use blanks within a key and value pair; a blank signals the end of one key/value pair.
- Because blanks separate key and value pairs, you must be careful when a value you supply contains blanks. When you specify a value that contains blanks, enclose the value in single quotes or double quotes. Because some shells processes quotes differently, you might have to do some testing to determine whether you must use single or double quotes.

  Example: Suppose you invoke the scripting tool, passing this input:

  ```
  /WebSphere/V5R0M0/AppServer/bin:>acrct -earfile
  usr/lpp/myapps/applclient2.ear -define
  -provider providername='WebSphere JMS Provider'
  ```

The response is an error message along with an echo of the input string that the shell receives, followed by the input string as the scripting tool will process it:

```
-earfile usr/lpp/myapps/applclient2.ear -define
-provider providername=WebSphere JMS Provider
String to be parsed by the Scripting Tool:

-earfile usr/lpp/myapps/applclient2.ear -define
-provider providername=WebSphere JMS Provider


Ear file is missing or is improperly specified.
Invalid syntax:
 -earfile usr/lpp/myapps/applclient2.ear -define -provider providername=
   WebSphere JMS Provider
```

As you can see from the response, the shell has stripped off the single quotes, and passes invalid input to the scripting tool. To correct the problem, you need to use double quotes.

- The number of key and value pairs you specify depends on the type of resource or provider you are configuring. For each resource to be configured, use this information to determine which resource or provider properties are required.

The following examples demonstrate correct syntax:

Defining a new provider for an application client, using an input file:

```
acrct -earfile usr/lpp/myapps/applclient1.ear -define -provider -f
usr/lpp/myapps/inputProvider1.def
```

Defining a new provider for an application client, specifying properties directly on the command line:

```
acrct -earfile usr/lpp/myapps/applclient2.ear -define -provider -p
providertype=DataSourceProvider name=DB2UDBV7
```

Defining a new provider and deleting the resource it replaces, in the same EAR file:

```
acrct -earfile usr/lpp/myapps/applclient1.ear -define -provider -f
usr/lpp/myapps/inputProvider2.def -delete -resource -f
usr/lpp/myapps/inputProvider1.def
```

Defining a new provider in more than one EAR file:

```
acrct -earfile usr/lpp/myapps/applclient1.ear -define -provider -f
usr/lpp/myapps/inputProvider2.def -earfile
usr/lpp/myapps/applclient2.ear -define -provider -f
usr/lpp/myapps/inputProvider2.def
```

Determining required properties for z/OS application client resources

When you deploy application clients on z/OS or OS/390, you need to determine the required properties to specify when using the z/OS scripting version of the Application Client Resource Configuration Tool (ACRCT).

Note: This procedure applies only for J2EE application clients.

Use the following information to determine required properties to specify for application client resources.

If the application uses this type of resource or provider:	Find required and optional properties in these articles:
Data source	• Datasource Provider • Datasource

If the application uses this type of resource or provider:	Find required and optional properties in these articles:
JMS	• JMS Provider • JMS Connection • JMS Destination
Mail session	• Mail provider • Mail session
Resource environment	• Resource environment provider • Resource environment entry
URL	• URL provider • URL factory
WebSphere MQ queue	• WebSphere MQ queue connection factory • WebSphere MQ queue destination factory
WebSphere MQ topic	• WebSphere MQ topic connection factory • WebSphere MQ topic destination factory
WebSphere queue	• WebSphere queue connection factory • WebSphere queue destination factory
WebSphere topic	• WebSphere topic connection factory • WebSphere topic destination factory

Properties for data source providers

name	required
description	optional
implementation	optional
classpath	optional

Properties for data sources

providertype=DataSourceProvider	required
providername	required
name	required
description	optional
jndiname	required
databasename	optional
user	optional
password	optional

providertype=DataSourceProvider
providername=DB2UDBV7
name=″PolicyDatasource″
description=″Datasource for Policy App″
jndiname=jdbc/PolicyDS
databasename=POLICYAPP
user=dbuser
password=dbpw

Properties for JMS providers

name	required
description	optional
classpath	optional
externalinitialcontextfactory	optional
externalproviderurl	optional

Properties for JMS connections

providertype=JMSProvider	required	
providername	required	
name	required	
description	optional	
jndiname	required	
externaljndiname	optional	
type	required	Valid values are: • **QUEUE** • **TOPIC**
user	optional	
password	optional	

Properties for JMS destinations

providertype=JMSProvider	required	
providername	required	
name	required	
description	optional	
jndiname	required	
externaljndiname	optional	
type	required	Valid values are: • **QUEUE** • **TOPIC**

Properties for mail providers

name	required
description	optional
classpath	optional

Properties for mail sessions

providertype=MailProvider	required	
providername	required	
name	required	

description	optional	
jndiname	required	
mailfrom	optional	
mailstorehost	optional	
mailstoreuser	optional	
mailstorepassword	optional	
mailtransporthost	optional	
mailtransportprotocol	required	Valid values are: • **smtp** • **imap** • **pop3**
mailtransportuser	optional	
mailtransportpassword	optional	
debug	required	Valid values are **True** or **False**.

Properties for resource environment providers

name	required
description	optional
classpath	optional

Properties for resource environment entries

providertype=ResourceEnvironmentProvider	required
providername	required
name	required
description	optional
jndiname	required

Properties for URL providers

name	required
description	optional
protocol	optional
classpath	optional
streamhandlerclass	optional

Properties for URL factories

providertype=URLProvider	required
providername	required
name	required
description	optional

jndiname	required
url	required

Properties for WebSphere MQ queue connection factories

providertype=JMSProvider	required	
providername="MQ JMS Provider"	required	
name	required	
description	optional	
jndiname	required	
transporttype	required	Valid values are: • **CLIENT** • **BINDINGS**
clientid	optional	
user	optional	
password	optional	
channel	optional	
ccsid	optional	

Properties for WebSphere MQ queue destination factories

providertype=JMSProvider	required	
providername="MQ JMS Provider"	required	
name	required	
description	optional	
jndiname	required	
persistence	required	Valid values are: • **APPLICATION_DEFINED** • **QUEUE_DEFINED** • **PERSISTENT** • **NONPERSISTENT**
priority	required	Valid values are: • **APPLICATION_DEFINED** • **QUEUE_DEFINED** • ***specified_integer*** **from 0 through 9**
expiry	required	Valid values are: • **APPLICATION_DEFINED** • **UNLIMITED** • ***specified_value***
basequeuename	required	
basequeuemanagername	optional	

targetclient	required	Valid values are: • **JMS** • **MQ**
ccsid	optional	
usenativeencoding	required	Valid values are: • **True** • **False**

Properties for WebSphere MQ topic connection factories

providertype=JMSProvider	required	
providername=″MQ JMS Provider″	required	
name	required	
description	optional	
jndiname	required	
transporttype	required	Valid values are: • **CLIENT** • **BINDINGS**
clientid	optional	
brokercontrolqueue	optional	
brokerqueuemanager	optional	
brokerpubqueue	optional	
brokersubqueue	optional	
brokerccsubq	optional	
brokerversion	required	Valid values are: • **MA0C** • **MQSI**
userid	optional	
password	optional	
ccsid	optional	
channel	optional	

Properties for WebSphere MQ topic destination factories

providertype=JMSProvider	required	
providername=″MQ JMS Provider″	required	
name	required	
description	optional	
jndiname	required	

persistence	required	Valid values are: • **APPLICATION_DEFINED** • **QUEUE_DEFINED** • **PERSISTENT** • **NONPERSISTENT**
priority	required	Valid values are: • **APPLICATION_DEFINED** • **QUEUE_DEFINED** • ***specified_integer*** from 0 through 9
expiry	required	Valid values are: • **APPLICATION_DEFINED** • **UNLIMITED** • ***specified_value***
basetopicname	required	
targetclient	required	Valid values are: • **JMS** • **MQ**
brokerdursubqueue	optional	
brokerccdursubqueue	optional	
ccsid	optional	
usenativeencoding	required	Valid values are: • **True** • **False**

Properties for WebSphere queue connection factories

providertype=JMSProvider	required
providername="WebSphere JMS Provider"	required
name	required
description	optional
jndiname	required
node	required
servername	required
user	optional
password	optional

Properties for WebSphere queue destination factories

providertype=JMSProvider	required	
providername="WebSphere JMS Provider"	required	
name	required	
description	optional	
jndiname	required	

node	required	
persistence	required	Valid values are: • **APPLICATION_DEFINED** • **PERSISTENT** • **NONPERSISTENT**
priority	required	Valid values are: • **APPLICATION_DEFINED** • ***specified_integer*** from 0 through 9
expiry	required	Valid values are: • **APPLICATION_DEFINED** • **UNLIMITED** • ***specified_value***

Properties for WebSphere topic connection factories

providertype=JMSProvider	required	
providername="WebSphere JMS Provider"	required	
name	required	
description	optional	
jndiname	required	
servername	required	
node	required	
port	required	Valid values are: • **QUEUED** • **DIRECT**
clientid	optional	
userid	optional	
password	optional	

Properties for WebSphere topic destination factories

providertype=JMSProvider	required	
providername="WebSphere JMS Provider"	required	
name	required	
description	optional	
jndiname	required	
topic	required	
persistence	required	Valid values are: • **APPLICATION_DEFINED** • **PERSISTENT** • **NONPERSISTENT**

priority	required	Valid values are: • **APPLICATION_DEFINED** • ***specified_integer*** from 0 through 9
expiry	required	Valid values are: • **APPLICATION_DEFINED** • **UNLIMITED** • ***specified_value***

Deploying application clients on workstation platforms

After developing an application client, deploy this application on client machines. *Deployment* consists of pulling together the various artifacts that the application client requires.

If you plan to deploy the client on z/OS or OS/390, you have two options:
- Run the Application Client Resource Configuration Tool (ACRCT) on Windows, or
- Run the Client Container Resource Configuration Scripting tool on z/OS.

Both of these tools produce equivalent output. They both provide resource definitions for an application client, which are stored in the application client `.ear` file. The application client run time (or container) uses these configurations for resolving and creating an instance of the resources for the application client.

Note: This task only applies to J2EE application clients. Only perform this task if you configured your J2EE application client to use resource references.

1. Start the ACRCT and open an EAR file.
2. Configure new data source providers.
3. Configure mail providers and sessions.
4. Configure URL providers and sessions.
5. Configure Java messaging client resources.
6. Configure new environment entries.
7. (Optional) Remove application client resources.
8. Save the EAR file.

Starting the Application Client Resource Configuration Tool and opening an EAR file

Note: This task only applies to J2EE application clients.

1. Open a command prompt and change to the `install_root\bin` directory.
2. Run the `clientConfig.bat` file for a Windows system .
3. Open an EAR file within the Application Client Resource Configuration Tool (ACRCT):
   - Click **File** > **Open**.
   - Select the file and click **Open**.
4. Save your changes to the file and close the tool:
   - Click **File** > **Save**.
   - Click **File** > **Exit**.

Data sources for application clients

The J2EE application client does not support looking up or directly accessing data sources configured on WebSphere Application Server because the J2EE application client does not support Java 2 Connection Factories. To use a data source directly from the client application, you must use the ACRCT to configure your data source. In addition, WebSphere Application Server and WebSphere Application Server clients do not provide client database drivers to be used directly from a J2EE application client. If your application client accesses a database directly, you must provide the database drivers on the client machine. You might contact your database vendor to acquire client database driver code and licenses. Instead of accessing the database directly, it is recommended that your client application use an enterprise bean. Accessing a database through an enterprise bean eliminates the need to have database drivers on the client machine, since the database access is handled by the enterprise bean running on the WebSphere Application Server. For a current list of providers that are supported on the WebSphere Application Server go the following site:

Supported hardware, software, and APIs

Configuring new data source providers (JDBC providers) for application clients

During this task, you create new data source providers, also known as JDBC providers, for your application client. In a separate administrative task, install the Java code for the required data source provider on the client machine on which the application client resides.

1. Start the tool and open the EAR file for which you want to configure the new data source provider. The EAR file contents display in a tree view.
2. Select the JAR file in which you want to configure the new data source provider from the tree.
3. Expand the JAR file to view its contents.
4. Click the **Data Source Providers** folder. Do one of the following:
   - Right-click the folder and click **New Provider**.
   - Click **Edit > New** on the menu bar.
5. Configure the data source provider properties in the resulting property dialog.
6. Click **OK** when you finish.
7. Click **File > Save** on the menu bar to save your changes.

Configuring new data source providers

During this task, you will create new data source providers, also known as JDBC drivers, for your application client. In a separate administrative task, install the Java code for the required data source provider on the client machine where the application client resides.

1. Start the ACRCT, click **File > Open**, and select the EAR file for which you want to configure the new data source provider. The EAR file contents display in a tree view.
2. Select the JAR file in which you want to configure the new data source provider from the tree.
3. Expand the JAR file to view its contents.
4. Right click the **Data Source Providers** folder and select **New Provider**.
5. Configure the data source provider properties in the resulting property dialog.

6. Click **OK**.
7. Click **File** > **Save** to save your changes.

Example: Configuring data source provider and data source settings: The purpose of this article is to help you to configure data source provider and data source settings.

- Required fields:
  - Data Source Provider Properties page: name
  - Data Source Properties page: name, jndiName
- Special cases:
  - The user name and password fields have no equivalant xmi tags. You must specify these fields in the custom properties.
  - The password is encrypted when you use the Application Client Resource Configuration Tool (ACRCT). If you do not use the ACRCT, the field cannot be encrypted.
- Example:

```
<resources.jdbc:JDBCProvider xmi:id="JDBCProvider_1" name="jdbcProvider:name"
description="jdbcProvider:description" implementationClassName="jdbcProvider:
ImplementationClass">
<classpath>jdbcProvider:classPath</classpath>
<factories xmi:type="resources.jdbc:WAS40DataSource" xmi:id="WAS40DataSource_1"
name="jdbcFactory:name" jndiName="jdbcFactory:jndiName"
description="jdbcFactory:description" databaseName="jdbcFactory:databasename">
<propertySet xmi:id="J2EEResourcePropertySet_13">
<resourceProperties xmi:id="J2EEResourceProperty_13" name="jdbcFactory:customName"
value="jdbcFactory:customValue"/>
<resourceProperties xmi:id="J2EEResourceProperty_14" name="user"
value="jdbcFactory:user"/>
<resourceProperties xmi:id="J2EEResourceProperty_15" name="password"
value="{xor}NTs9PBk+PCswLSZlMT4yOg=="/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_14">
<resourceProperties xmi:id="J2EEResourceProperty_16" name="jdbcProvider:customName"
value="jdbcProvider:customeValue"/>
</propertySet>
</resources.jdbc:JDBCProvider>
```

Data source provider settings for application clients:

Use this page to create a data source under a JDBC provider which provides the specific JDBC driver implementation class.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Data Source Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the display name for the data source.

For example you can set this field to *Test Data Source*.

Data type	String

Description:

Specifies a text description for the resource.

Data type	String

Class Path:

A list of paths or jarfile names which together form the location for the resource provider classes.

Implementation class:

Use this setting to perform database specific functions.

Data type	String
Default	Dependent on JDBC driver implementation class

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Data source properties for application clients:

Use this page to create or modify the data sources.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Data Source Providers** > *Data source provider instance*. Right-click **Data Sources** and click **New**. The following fields are displayed on the **General** tab:

Name:

Specifies the display name of this data source.

Data type	String

Description:

Specifies a text description of the data source.

Data type	String

JNDI Name:

The application client run time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Assembly Tool.

Database Name:

The name of the database to which you want to connect.

User:

Use the user ID with the Password property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the User ID property, you must also specify a value for the Password property. The connection factory User ID and Password properties are used if the calling application does not provide a userid and password explicitly.

Password:

Use the password with the User ID property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the User ID property, you must also specify a value for the Password property.

Re-Enter Password:

Confirms the password.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new data sources for application clients

During this task, you create new data sources for your application client.

1. Click the data source provider for which you want to create a data source in the tree. Do one of the following:
   - Configure a new data source provider.
   - Click an existing data source provider.
2. Expand the data source provider to view its **Data Sources** folder.
3. Click the folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the data source properties in the resulting property dialog.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring mail providers and sessions for application clients

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of JavaMail sessions and providers for your application clients to use.

1. Open the ACRCT.
2. Open an EAR file.
3. Locate the JavaMail objects in the tree that displays. For example, if your file contains JavaMail sessions, expand **Resources** > ***application*.jar** > **JavaMail Providers** > *java_mail_provider_instance* > **JavaMail Sessions**.

   In this example, ***java_mail_provider_instance*** is a particular JavaMail provider.

The JavaMail session instances are located in the **JavaMail Sessions** folder.

Mail provider settings for application clients

Use this page to implement the JavaMail API and create mail sessions.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Mail Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

The name of the JavaMail resource provider.

Description:

An optional description for the resource provider.

Class Path:

Specifies a list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Specifies the name of the protocol.

Classname:

Specifies the name of the class implementing the protocol. Leave this field blank if you want to use the default implementation.

Type:

This menu contains the following two values: `TRANSPORT` or `STORE`.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property.

The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Mail session settings for application clients

Use this page to configure mail session properties.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Mail Providers** > *mail provider instance*. Right-click **Mail Sessions** and click **New**. The following fields appear on the **General** tab:

Name:

Represents the administrative name of the JavaMail session object.

Description:

Provides an optional description for your administrative records.

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Mail Transport Host:

Specifies the server to connect to when sending mail.

Mail Transport Protocol:

Specifies the transport protocol to use when sending mail.

Mail Transport User:

Specifies the user ID to use when the mail transport host requires authentication.

Mail Transport Password:

Specifies the password to use when the mail transport host requires authentication.

Re-Enter Password:

Confirms the password.

Mail From:

Specifies the mail originator.

Mail Store Host:

Mail account host (or "domain") name.

Mail Store User:

The user ID of the mail account.

Mail Store Password:

The password of the mail account.

Re-Enter Password:

Confirms the password.

Mail Store Protocol:

Specifies the protocol to be used when receiving mail.

Mail Debug:

When true, JavaMail interaction with mail servers, along with these mail session properties will be printed to stdout.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring JavaMail provider and JavaMail session settings for application clients

The purpose of this article is to help you configure JavaMail provider and JavaMail session settings.

- Required fields:
  - JavaMail Provider Properties page: name, and at least one protocol provider
  - JavaMail Session Properties page: name, jndiName, mail transport protocol, mail store protocol
- Special cases:
  - The password is encrypted when using the ACRCT tool. Without the tool, you cannot encrypt this field.
- Example:

```
<resources.mail:MailProvider xmi:id="MailProvider_1" name="Default Mail Provider"
description="IBM JavaMail Implementation">
<classpath>mailProvider:classpath</classpath>
<factories xmi:type="resources.mail:MailSession" xmi:id="MailSession_1"
name="mailSession:name" jndiName="mailSession:jndiName"
description="mailSession:description" mailTransportHost="mailSession:mailTransportHost"
mailTransportUser="mailSession:mailTransportUser"
mailTransportPassword="{xor}Mj42Mww6LCw2MDFlMT4yOg=="
mailFrom="mailSession:mailFrom" mailStoreHost="mailSession:mailStoreHost"
mailStoreUser="mailSession:mailStoreUser"
mailStorePassword="{xor}Mj42Mww6LCw2MDFlMT4yOg==" debug="true"
mailTransportProtocol="ProtocolProvider_1" mailStoreProvider="ProtocolProvider_1">
<propertySet xmi:id="J2EEResourcePropertySet_1">
<resourceProperties xmi:id="J2EEResourceProperty_1"
name="mailSession:customName" value="mailSession:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_2">
<resourceProperties xmi:id="J2EEResourceProperty_2" name="mailProvider:customName"
value="mailProvider:customValue"/>
</propertySet>
<protocolProviders xmi:id="ProtocolProvider_1" protocol="smtp"
classname="smtp:className"/>
```

```
<protocolProviders xmi:id="ProtocolProvider_2" protocol="pop3"
classname="pop3:className"/>
<protocolProviders xmi:id="ProtocolProvider_3" protocol="imap"
classname="imap:className"/>
</resources.mail:MailProvider>
```

Configuring new mail sessions for application clients

During this task, you configure new mail sessions for your application client. The mail sessions are associated with the pre-configured default mail provider supplied by the product.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file. The EAR file contents are displayed in a tree view.
2. Select the JAR file in which you want to configure the new JavaMail session.
3. Expand the JAR file to view its contents.
4. Click **JavaMail Providers** > **MailProvider** > **JavaMail Sessions**. Complete one of the following actions:
   - Right-click the **JavaMail Sessions** folder and select **New Factory**.
   - Click **Edit** > **New** on the menu bar.
5. Configure the JavaMail session properties in the resulting property dialog.
6. Click **OK**.
7. Click **File** > **Save** on the menu bar to save your changes.

URLs for application clients

A *Uniform Resource Locator* (URL) is an identifier that points to an electronically accessible resource, such as a directory file on a machine in a network, or a document stored in a database.

URLs appear in the format *scheme*:*scheme_information*.

You can represent a *scheme* as `http`, `ftp`, `file`, or another term that identifies the type of resource and the mechanism by which you can access the resource.

In a World Wide Web browser location or address box, a URL for a file available using HyperText Transfer Protocol (HTTP) starts with `http:`. An example is `http://www.ibm.com`. Files available using File Transfer Protocol (FTP) start with `ftp:`. Files available locally start with `file:`.

The *scheme_information* commonly identifies the Internet machine making a resource available, the path to that resource, and the resource name. The scheme_information for HTTP, FTP and File generally starts with two slashes (//), then provides the Internet address separated from the resource path name with one slash (/). For example,

`http://www-4.ibm.com/software/webservers/appserv/library.html`.

For HTTP and FTP, the path name ends in a slash when the URL points to a directory. In such cases, the server generally returns the default index for the directory.

URL providers for the Application Client Resource Configuration Tool

A URL provider implements the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP). This provider, comprised of a pair of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Configuring new URL providers for application clients

During this task, you create URL providers and URLs for your client application. In a separate administrative task, you must install the Java code for the required URL provider on the client machine on which the client application resides.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new URL provider. The EAR file contents display in a tree view.
3. Select the JAR file in which you want to configure the new URL provider from the tree.
4. Expand the JAR file to view its contents.
5. Click the folder called **URL Providers**. Complete one of the following actions:
   - Right-click the folder and click **New Provider**.
   - Click **Edit** > **New** on the menu bar.
6. Configure the URL provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save** on the menu bar to save your changes.

Configuring URL providers and sessions using the Application Client Resource Configuration Tool

Use the Application Client Resource Configuration Tool (ACRCT) to edit the configurations of URL providers and URLs to be used by your application clients.

1. Open the ACRCT.
2. Open an EAR file.
3. Locate the URL objects in the tree that displays. For example, if your file contains URL providers and URLs, expand **Resources** -> ***application*.jar** -> **URL Providers** -> ***url_provider_instance***

   where ***url_provider_instance*** is a particular URL provider.
4. If you expand the tree further, you will also see the **URLs** folders containing the URL instances for each URL provider instance.

URL settings for application clients:

Use this page to implement the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP).

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **URL Providers** > *URL provider instance*. Right-click **URLs** and click **New**. The following fields appear on the **General** tab.

This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

Administrative name for the URL

Description:

Optional description of the URL, for your administrative records

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

URL:

A Uniform Resource Locator (URL) name that points to an internet or intranet resource. For example: `http://www.ibm.com`

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

URL provider settings for application clients:

Use this page create new URLs.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **URL Providers** > and click **New**. The following fields appear on the **General** tab.

A URL provider implements the function for a particular URL protocol, such as Hyper Text Transfer Protocol (HTTP). This provider, comprised of classes, extends the `java.net.URLStreamHandler` and `java.net.URLConnection` classes.

Name:

Administrative name for the URL

Description:

Optional description of the URL, for your administrative records

Class Path:

A list of paths or JAR file names which together form the location for the resource provider classes.

Protocol:

Protocol supported by this stream handler. For example, "nntp", "smtp", "ftp", and so on.

To use the default protocol, leave this field blank.

Stream handler class:

Fully qualified name of a User-defined Java class that extends `java.net.URLStreamHandler` for a particular URL protocol, such as FTP.

To use the default stream handler, leave this field blank.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring URL and URL provider settings for application clients

The purpose of this article is to help you to configure URL and URL provider settings.

- Required fields:
  - URL Properties page: name, jndiName, url
  - URL Provider Properties page: name
- Example:

```
<resources.url:URLProvider xmi:id="URLProvider_1" name="urlProvider:name"
description="urlProvider:description"
streamHandlerClassName="urlProvider:streamHandlerClass"
protocol="urlProvider:protocol">
<classpath>urlProvider:classpath</classpath>
<factories xmi:type="resources.url:URL" xmi:id="URL_1" name="urlFactory:name"
jndiName="urlFactory:jndiName" description="urlFactory:description"
spec="urlFactory:url">
<propertySet xmi:id="J2EEResourcePropertySet_18">
<resourceProperties xmi:id="J2EEResourceProperty_20" name="urlFactory:customName"
value="urlFactory:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_19">
<resourceProperties xmi:id="J2EEResourceProperty_21" name="urlProvider:customName"
value="urlProvider:customValue"/>
</propertySet>
</resources.url:URLProvider>
```

Configuring new URLs with the Application Client Resource Configuration Tool

During this task, you create URLs for your client application.

1. Click the URL provider for which you want to create a URL in the tree. Do one of the following:
   - Configure a new URL provider.
   - Click an existing URL provider.
2. Expand the URL provider to view the **URLs** folder.
3. Click the folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit -> New** on the menu bar.

4. Configure the URL properties in the resulting property dialog.
5. Click **OK** when you finish.
6. Click **File** -> **Save** in the menu bar to save your changes.

WebSphere asynchronous messaging using the Java Message Service API for the Application Client Resource Configuration Tool

WebSphere Application Server supports asynchronous messaging as a method of communication based on the Java Message Service (JMS) programming interface. The JMS interface provides a common way for Java programs (clients and J2EE applications) to create, send, receive, and read asynchronous requests as JMS messages.

This topic provides an overview of asynchronous messaging using JMS support provided by the WebSphere Application Server.

The base support for asynchronous messaging using the JMS API provides the common set of JMS interfaces and associated semantics that define how a JMS client can access the facilities of a JMS provider. This support enables WebSphere J2EE applications, as JMS clients, to exchange messages asynchronously with other JMS clients, by using JMS destinations (queues or topics). A J2EE application can use JMS queue destinations for point-to-point messaging and JMS topic destinations for Pub and Sub messaging. A J2EE application can explicitly poll for messages on a destination then retrieve messages for processing by business logic beans (enterprise beans).

With the base JMS/XA support, the J2EE application uses standard JMS calls to process messages, including any responses or outbound messaging. An enterprise bean can handle responses acting as a sender bean, or within the enterprise bean that receives the incoming messages. Optionally, this process can use two-phase commit within the scope of a transaction. This level of function for asynchronous messaging is called *bean-managed messaging*, and gives an enterprise bean complete control over the messaging infrastructure; for example, connection and session pool management. The common container has no role in *bean-managed messaging*.

WebSphere Application Server also supports automatic asynchronous messaging using message-driven beans (a type of enterprise bean defined in the EJB 2.0 specification) and JMS listeners (part of the JMS application server facilities). Messages are automatically retrieved from JMS destinations, optionally within a transaction, then sent to the message-driven bean in a J2EE application, without the application having to explicitly poll JMS destinations.

Configuring Java messaging client resources

In a separate administrative task, install the Java Message Service (JMS) client on the client machine where the application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

During this task, you create new JMS provider configurations for your application client. The application client can use a messaging service through the Java Message Service APIs. A JMS provider provides two kinds of J2EE factories. One is a *JMS connection factory*, and the other is a *JMS destination factory*.

1. Start the Application Client Resource Configuration Tool (ACRCT).

2. Open the EAR file for which you want to configure the new JMS provider. The EAR file contents are in the displayed tree view.
3. Select the JAR file in which you want to configure the new JMS provider from the tree.
4. Expand the JAR file to view its contents.
5. Click the **JMS Providers** folder and click **New Provider**.
6. Configure the JMS provider properties in the resulting property dialog.
7. Click **OK**.
8. Click **File** > **Save**.

Configuring new JMS providers with the Application Client Resource Configuration Tool

During this task, you will create new JMS provider configurations for your application client. The application client can make use of a messaging service through the Java Message Service APIs. A JMS provider provides two kinds of J2EE factories. One is a JMS Connection factory, and the other is a JMS destination factory.

In a separate administrative task, you must install the JMS client on the client machine where the application client resides. The messaging product vendor must provide an implementation of the JMS client. For more information, see your messaging product documentation.

1. Start the tool and open the EAR file for which you want to configure the new JMS provider. The EAR file contents will be displayed in a tree view.
2. From the tree, select the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Click the folder called **JMS Providers**. Do one of the following:
   - Right-click the folder and select **New Provider**.
   - On the menu bar, click **Edit -> New**.
5. In the resulting property dialog, configure the JMS provider properties.
6. When finished, click **OK**.
7. On the menu bar, click **File -> Save** to save your changes.

JMS provider settings for application clients

Use this page to configure properties of the JMS provider, if you want to use a JMS provider other than the internal WebSphere JMS provider or the MQSeries JMS provider.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **JMS Providers** > click **New**. The following fields appear on the **General** tab.

Name:

The name by which the JMS provider is known for administrative purposes.

Data type	String

Description:

A description of the JMS provider, for administrative purposes

Data type	String

Class Path:

A list of paths or jarfile names which together form the location for the resource provider classes.

Context factory class:

The Java classname of the initial context factory for the JMS provider.

For example, for an LDAP service provider the value has the form: `com.sun.jndi.ldap.LdapCtxFactory`.

Data type	String

Provider URL:

The JMS provider URL for external JNDI lookups.

For example, an LDAP URL for a JMS provider has the form: `ldap://hostname.company.com/contextName`.

Data type	String

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere queue connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected queue connection factory for use with the internal WebSphere JMS provider that is installed with WebSphere Application Server. These configuration properties control how connections are created to the associated JMS queue destination.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **WAS Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

A queue connection factory is used to create JMS connections to queue destinations. The queue connection factory is created by the internal WebSphere JMS provider. A queue connection factory for the internal WebSphere JMS provider has the following properties:

Name:

The name by which this queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type	String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

The User ID used, with the **Password** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a User ID and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the `userid` and `password` to the JMS server.

Data type	String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a `userid` and `password` explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Re-Enter Password:

Confirms the password.

Node:

The WebSphere node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type	String

Application Server:

Enter the name of the application server. This name is not the host name of the machine, but the name of the configured application server.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at run time for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere topic connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected topic connection factory for use with the internal WebSphere JMS provider. These configuration properties control how connections are created to the associated JMS topic destination.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **WAS Topic Connection Factories** and click **New**. The following fields appear on the **General** tab.

A topic connection factory is used to create JMS connections to topic destinations. The topic connection factory is created by the associated JMS provider. A topic connection factory for the internal WebSphere JMS provider has the following properties.

Name:

The name by which this queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type	String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createTopicConnection(). The JMS client flows the userid and password to the JMS server.

Data type	String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type	String
Default	Null

Re-Enter Password:

Confirms the password.

Node:

The WebSphere node name of the administrative node where the JMS server runs for this connection factory. Connections created by this factory connect to that JMS server.

Data type	Enum
Default	Null
Range	Pull-down list of nodes in the WebSphere administrative domain.

Application Server:

Enter the name of the application server. This name is not the host name of the machine, but the name of the configured application server.

Port:

Which of the two ports that connections use to connect to the JMS Server. The QUEUED port is for full-function JMS publish/subscribe support, the DIRECT port is for non-persistent, non-transactional, non-durable subscriptions only.

Note: Message-driven beans cannot use the direct listener port for publish/subscribe support. Therefore, any topic connection factory configured with **Port** set to `Direct` cannot be used with message-driven beans.

Data type	Enum
Units	Not applicable
Default	QUEUED
Range	**QUEUED** The listener port used for full-function JMS-compliant, publish/subscribe support. **DIRECT** The listener port used for direct TCP/IP connection (non-transactional, non-persistent, and non-durable subscriptions only) for publish/subscribe support. The TCP/IP port numbers for these ports are defined on the WebSphere Internal JMS Server.

Client Id:

The JMS client identifier used for connections to the MQSeries queue manager.

Data type	String

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere queue destination settings for application clients

Use this panel to view or change the configuration properties of the selected queue destination for use with the WebSphere JMS provider.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **WAS Queue Destinations** and click **New**. The following fields appear on the **General** tab.

A queue destination is used to configure the properties of a JMS queue. Connections to the queue are created by the associated queue connection factory for the internal WebSphere JMS provider. A queue for use with the internal WebSphere JMS provider has the following properties.

Name:

The name by which the queue is known for administrative purposes within IBM WebSphere Application Server.

Data type	String

Description:

A description of the queue, for administrative purposes

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them onto the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the **Specified expiry** property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

WebSphere topic destination settings for application clients

Use this panel to view or change the configuration properties of the selected topic destination for use with the internal WebSphere JMS provider.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **WAS Topic Destinations** and click **New**. The following fields appear on the **General** tab.

A topic destination is used to configure the properties of a JMS topic for the associated JMS provider. Connections to the topic are created by the associated topic connection factory. A topic for use with the internal WebSphere JMS provider has the following properties.

Name:

The name by which the topic is known for administrative purposes.

Data type	String

Description:

A description of the topic, for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Topic Name: The name of the topic as defined to the JMS provider.

Data type	String

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them onto the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable

Default	APPLICATION_DEFINED
Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the **Specified expiry** property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

MQSeries queue connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected queue connection factory for use with the MQSeries JMS provider. These configuration properties control how connections are created to the associated JMS queue destination.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **MQ Queue Connection Factories** and click **New**. The following fields appear on the **General** tab.

A queue connection factory creates JMS connections to queue destinations. The queue connection factory is created by the MQSeries JMS provider. A queue connection factory for the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries for JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book, located in the WebSphere MQ Family library.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

Name:

The name by which this queue connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS connection factories across the WebSphere administrative domain.

Data type	String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the userid and password to the JMS server.

Data type	String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type	String
Default	Null

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the MQSeries queue manager for this connection factory.

Connections created by this factory connect to that queue manager.

Data type	String

Host:

The name of the host on which the WebSphere MQ queue manager runs, for client connection only.

Data type	String
Default	Null
Range	A valid TCP/IP hostname

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Data type	Integer
Default	Null
Range	A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Channel:

The name of the channel used for connection to the WebSphere MQ queue manager, for client connection only.

Data type	String
Default	Null
Range	1 through 20 ASCII characters

Transport type:

Specifies whether the WebSphere MQ client connection or JNI bindings are used for connection to the WebSphere MQ queue manager. The external JMS provider controls the communication protocols between JMS clients and JMS servers. Tune the transport type when you are using non-ASF nonpersistent, nondurable, nontransactional messaging or when you want to satisfy security issues and the client is local to the queue manager node.

Data type	Enum

Units
Not applicable

Default
BINDINGS

Range

BINDINGS
JNI bindings are used to connect to the queue manager. BINDINGS is a shared memory protocol and can only be used when the queue manager is on the same node as the JMS client and comes at some security risks that should be addressed through the use of EJB roles.

CLIENT
WebSphere MQ client connection is used to connect to the queue manager. CLIENT is a typical TCP-based protocol.

DIRECT
For WebSphere MQ Event Broker using DIRECT mode. DIRECT is a lightweight sockets protocol used in nontransactional, nondurable and nonpersistent Publish/Subscribe messasging. DIRECT is only works for clients and message-driven beans using the non-ASF protocol.

QUEUED
QUEUED is a standard TCP protocol.

Recommended

Queue connection factory transport type
BINDINGS is faster by 30% or more, but it lacks security. When you have security concerns, BINDINGS is more desirable than CLIENT.

Topic connection factory transport type
DIRECT is the fastest and should be used where possible. Use BINDINGS when you want to satisfy additional security tasks and the queue manager is local to the JMS client. QUEUED is fallback for all other cases. Note, WebSphere MQ 5.3 before CSD2 with the DIRECT setting can lose messages when used with message-driven beans and under load. This also happens with client-side based applications unless the broker's maxClientQueueSize is set to 0. You can set this to 0 with the command `#wempschangeproperties WAS_nodeName_server1 -e default -o DynamicSubscriptionEngine -n maxClientQueueSize -v 0 -x executionGroupUUID`, where executionGroupUUID can be found by starting the broker and looking in the Event Log/Applications for event 2201. This value is usually ffffffff-0000-0000-000000000000.

Client ID:

The JMS client identifier used for connections to the MQSeries queue manager.

Data type	String

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type	String

For more information about supported CCSIDs, and about converting between message data from one coded character set to another, see the *WebSphere MQ System Administration* and the *WebSphere MQ Application Programming Reference* books. These are available from the WebSphere MQ messaging multiplatform and platform-specific books Web pages; for example, at http://www-3.ibm.com/software/ts/mqseries/library/manualsa/manuals/platspecific.html, the IBM Publications Center, or from the WebSphere MQ collection kit, SK2T-0730.

Message Retention:

Select this check box to specify that unwanted messages are to be left on the queue. Otherwise, unwanted messages are dealt with according to their disposition options.

Data type	Enum
Units	Not applicable
Default	Cleared
Range	**Selected** Unwanted messages are left on the queue. **Cleared** Unwanted messages are dealt with according to their disposition options.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

MQSeries topic connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected topic connection factory for use with the MQSeries JMS provider. These configuration properties control how connections are created to the associated JMS topic destination.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **MQ Topic Connection Factories** and click **New**.

A topic connection factory is used to create JMS connections to topic destinations. The topic connection factory is created by the MQSeries JMS provider. A topic connection factory for the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

Name:

The name by which this topic connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the JMS provider.

Data type	String

Description:

A description of this topic connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The JNDI name that is used to bind the topic connection factory into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 45 ASCII characters

User:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User** property, you must also specify a value for the **Password** property.

The connection factory **User** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createTopicConnection(). The JMS client flows the userid and password to the JMS server.

Data type	String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type	String
Default	Null

Re-Enter Password:

Confirms the password.

Queue Manager:

The name of the MQSeries queue manager for this connection factory. Connections created by this factory connect to that queue manager.

Data type	String

Host:

The name of the host on which the WebSphere MQ queue manager runs, for client connection only.

Data type	String
Default	Null
Range	A valid TCP/IP hostname

Port:

The TCP/IP port number used for connection to the WebSphere MQ queue manager, for client connection only.

This port must be configured on the WebSphere MQ queue manager.

Data type	Integer
Default	Null
Range	A valid TCP/IP port number, configured on the WebSphere MQ queue manager.

Channel:

The name of the channel used for connection to the WebSphere MQ queue manager, for client connection only.

Data type	String
Default	Null
Range	1 through 20 ASCII characters

Transport Type:

Whether MQSeries client connection or JNDI bindings are used for connection to the MQSeries queue manager.

Data type	Enum
Units	Not applicable
Default	BINDINGS
Range	**CLIENT** MQSeries client connection is used to connect to the MQSeries queue manager. **BINDINGS** JNDI bindings are used to connect to the MQSeries queue manager.

Client Id:

The JMS client identifier used for connections to the MQSeries queue manager.

Data type	String

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type	String

Broker Control Queue:

The name of the broker control queue, to which all command messages (except publications and requests to delete publications) are sent

The name of the broker control queue. Publisher and subscriber applications, and other brokers, send all command messages (except publications and requests to

delete publications) to this queue.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Queue Manager:

The name of the MQSeries queue manager that provides the Pub/Sub message broker.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Pub Queue:

The name of the broker's input queue that receives all publication messages for the default stream

The name of the broker's input queue (stream queue) that receives all publication messages for the default stream. Applications can also send requests to delete publications on the default stream to this queue.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Sub Queue:

The name of the broker queue from which non-durable subscription messages are retrieved

The name of the broker's queue from which non-durable subscription messages are retrieved. The subscriber specifies the name of the queue when it registers a subscription.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker CCSubQ:

The name of the broker's queue from which non-durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

The name of the broker queue from which non-durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Broker Version:

Specifies whether the message broker is provided by the MQSeries MA0C SupportPac or newer versions of WebSphere message broker products.

Data type	Enum
Units	Not applicable
Default	Advanced
Range	**Advanced** The message broker is provided by newer versions of WebSphere message broker products (MQ Integrator and MQ Publish and Subscribe) **Basic** The message broker is provided by the MQSeries MA0C SupportPac (MQSeries - Publish/Subscribe)

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The accectable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

MQSeries queue destination settings for application clients

Use this panel to view or change the configuration properties of the selected queue destination for use with the MQSeries JMS provider.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **MQ Queue Destinations** and click **New**. The following fields appear on the **General** tab.

A queue destination configures the properties of a JMS queue. Connections to the queue are created by the associated queue connection factory for the MQSeries JMS provider. A queue for use with the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

Name:

The name by which the queue is known for administrative purposes within IBM WebSphere Application Server.

Data type	String

Description:

A description of the queue, for administrative purposes

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them onto the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Non persistent** Messages on the destination are not persistent.

Priority:

Whether the message priority for this destination is defined by the application or the **Specified priority** property

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The priority of messages on this destination is defined by the application that put them onto the destination. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Specified** The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the* ***Specified priority*** *property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type here the message priority for this queue, in the range 0 (lowest) through 9 (highest)

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type	Integer
Units	Message priority level
Default	Null
Range	0 (lowest priority) through 9 (highest priority)

Expiry:

Whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout)

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED

Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the **Specified expiry** property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type here the number of milliseconds (greater than 0) after which messages on this queue expire

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Base Queue Name:

The name of the queue to which messages are sent, on the queue manager specified by the **Base queue manager name** property

Data type	String

Base Queue Manager Name:

The name of the MQSeries queue manager to which messages are sent

This queue manager provides the queue specified by the **Base queue name** property.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	A valid MQSeries Queue Manager name, as 1 through 48 ASCII characters

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type String

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Data type Enum
Units Not applicable
Default NORMAL
Range
NORMAL
Normal integer encoding is used.
REVERSED
Reversed integer encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Data type Enum
Units Not applicable
Default NORMAL
Range
NORMAL
Normal decimal encoding is used.
REVERSED
Reversed decimal encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Floating point encoding:

If native encoding is not enabled, select the type of floating point encoding.

Data type Enum
Units Not applicable
Default IEEENORMAL
Range
IEEENORMAL
IEEE normal floating point encoding is used.
IEEEREVERSED
IEEE reversed floating point encoding is used.
S390 S390 floating point encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Select this checkbox to indicate that the queue destination should use native encoding (appropriate encoding values for the Java platform).

Data type	Enum
Units	Not applicable
Default	Cleared
Range	**Cleared** Native encoding is not used, so specify the properties below for integer, decimal, and floating point encoding. **Selected** Native encoding is used (to provide appropriate encoding values for the Java platform). For more information about encoding properties, see the MQSeries *Using Java* document.

Target client:

Whether the receiving application is JMS-compliant or is a traditional WebSphere MQ application

Data type	Enum
Units	Not applicable
Default	MQSeries
Range	**MQSeries** The target is a non-JMS, traditional WebSphere MQ application. **JMS** The target is a JMS-compliant application.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

MQSeries topic destination settings for application clients

Use this panel to view or change the configuration properties of the selected topic destination for use with the MQSeries JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *JMS provider instance*. Right-click **MQ Topic Destinations** and click **New**. The following fields appear on the **General** tab.

A topic destination is used to configure the properties of a JMS topic for the associated JMS provider. Connections to the topic are created by the associated topic connection factory. A topic for use with the MQSeries JMS provider has the following properties.

Note:

- The property values that you specify must match the values that you specified when configuring MQSeries JMS resources. For more information about configuring MQSeries JMS resources, see the MQSeries *Using Java* book.
- In MQSeries, names can have a maximum of 48 characters, with the exception of channels which have a maximum of 20 characters.

Name:

The name by which the topic is known for administrative purposes.

Data type	String

Description:

A description of the topic, for administrative purposes within IBM WebSphere Application Server.

JNDI Name:

The application client run time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

Persistence:

Specifies whether all messages sent to the destination are persistent, non-persistent, or have their persistence defined by the application.

Data type	Enum
Units	Not applicable
Default	APPLICATION_DEFINED
Range	**Application defined** Messages on the destination have their persistence defined by the application that put them onto the queue. **Queue defined** [WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties. **Persistent** Messages on the destination are persistent. **Non persistent** Messages on the destination are not persistent.

Priority:

Specifies whether the message priority for this destination is defined by the application or the **Specified priority** property.

Data type
Enum
Units
Not applicable
Default
APPLICATION_DEFINED
Range
Application defined
The priority of messages on this destination is defined by the application that put them onto the destination.
Queue defined
[WebSphere MQ destination only] Messages on the destination have their persistence defined by the WebSphere MQ queue definition properties.
Specified
The priority of messages on this destination is defined by the **Specified priority** property.*If you select this option, you must define a priority on the **Specified priority** property.*

Specified Priority:

If the **Priority** property is set to `Specified`, type the message priority for this queue, in the range 0 (lowest) through 9 (highest).

If the **Priority** property is set to `Specified`, messages sent to this queue have the priority value specified by this property.

Data type
Integer
Units
Message priority level
Default
Null
Range
0 (lowest priority) through 9 (highest priority)

Expiry:

Specifies whether the expiry timeout for this queue is defined by the application or the **Specified expiry** property, or messages on the queue never expire (have an unlimited expiry timeout).

Data type
Enum
Units
Not applicable
Default
APPLICATION_DEFINED

Range	**Application defined** The expiry timeout for messages on this queue is defined by the application that put them onto the queue. **Specified** The expiry timeout for messages on this queue is defined by the **Specified expiry** property.*If you select this option, you must define a timeout on the* ***Specified expiry*** *property.* **Unlimited** Messages on this queue have no expiry timeout, so those messages never expire.

Specified Expiry:

If the **Expiry timeout** property is set to `Specified`, type the number of milliseconds (greater than 0) after which messages on this queue expire.

Data type	Integer
Units	Milliseconds
Default	Null
Range	Greater than or equal to 0 • 0 indicates that messages never timeout • Other values are an integer number of milliseconds

Base Topic Name:

The name of the topic to which messages are sent

Data type	String

CCSID:

The coded character set identifier for use with the WebSphere MQ queue manager.

This coded character set identifier (CCSID) must be one of the CCSIDs supported by WebSphere MQ.

Data type	String

Integer encoding:

If native encoding is not enabled, select whether integer encoding is normal or reversed.

Data type	Enum
Units	Not applicable
Default	NORMAL

Range
: **NORMAL**
  Normal integer encoding is used.
: **REVERSED**
  Reversed integer encoding is used.

For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Decimal encoding:

If native encoding is not enabled, select whether decimal encoding is normal or reversed.

Data type	Enum
Units	Not applicable
Default	NORMAL
Range	**NORMAL** Normal decimal encoding is used. **REVERSED** Reversed decimal encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Floating point encoding:

If native encoding is not enabled, select the type of floating point encoding.

Data type	Enum
Units	Not applicable
Default	IEEENORMAL
Range	**IEEENORMAL** IEEE normal floating point encoding is used. **IEEEREVERSED** IEEE reversed floating point encoding is used. **S390** S390 floating point encoding is used. For more information about encoding properties, see the WebSphere MQ *Using Java* document.

Native encoding:

Select this check box to indicate that the queue destination should use native encoding (appropriate encoding values for the Java platform).

Data type	Enum
Units	Not applicable
Default	Cleared

Range	**Cleared** Native encoding is not used, so specify the properties above for integer, decimal, and floating point encoding. **Selected** Native encoding is used (to provide appropriate encoding values for the Java platform). For more information about encoding properties, see the MQSeries *Using Java* document.

BrokerDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved.

The name of the broker queue from which durable subscription messages are retrieved. The subscriber specifies the name of the queue when it registers a subscription.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

BrokerCCDurSubQueue:

The name of the broker queue from which durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

The name of the broker queue from which durable subscription messages are retrieved for a ConnectionConsumer. This property applies only for use of the Web container.

Data type	String
Units	En_US ASCII characters
Default	Null
Range	1 through 48 ASCII characters

Target Client:

Specifies whether the receiving application is JMS-compliant or is a traditional MQSeries application.

Data type	Enum
Units	Not applicable
Default	MQSeries

Range	**MQSeries** The target is a non-JMS, traditional MQSeries application. **JMS** The target is a JMS-compliant application.

Custom Properties:

Specifies the name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Generic JMS connection factory settings for application clients

Use this panel to view or change the configuration properties of the selected JMS connection factory for use with the associated JMS provider. These configuration properties control how connections are created to the associated JMS destination. To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *new JMS Provider instance*. Right click **JMS Connection Factories** > click **New**. The following fields appear on the **General** tab.

A JMS connection factory creates connections to JMS destinations. The JMS connection factory is created by the associated JMS provider. A JMS connection factory for a generic JMS provider (other than the internal WebSphere JMS provider or the MQSeries JMS provider) has the following properties:

Name:

The name by which this JMS connection factory is known for administrative purposes within IBM WebSphere Application Server. The name must be unique within the associated JMS provider.

Data type	String

Description:

A description of this connection factory for administrative purposes within IBM WebSphere Application Server.

Data type	String
Default	Null

JNDI Name:

The application client run-time uses this field to retrieve configuration information. The name must match the value of the **Name** field on the General tab in the Application Client Resource Reference section of the Application Assembly Tool.

User:

The user ID used, with the **Password** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

The connection factory **User ID** and **Password** properties are used if the calling application does not provide a userid and password explicitly; for example, if the calling application uses the method createQueueConnection(). The JMS client flows the userid and password to the JMS server.

Data type	String

Password:

The password used, with the **User ID** property, for authentication if the calling application does not provide a userid and password explicitly.

If you specify a value for the **User ID** property, you must also specify a value for the **Password** property.

Data type	String
Default	Null

Re-Enter Password:

Confirms the password entered in the Password field.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String

Connection Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for pub/sub).

Select one of the following options:

Queue
A JMS queue destination for point-to-point messaging.

Topic
A JMS topic destination for pub/sub messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Generic JMS destination settings for application clients

Use this panel to view or change the configuration properties of the selected JMS destination for use with the associated JMS provider.

To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **JMS Providers** > *new JMS Provider instance*. Right click **JMS Destinations** > click **New**. The following fields appear on the **General** tab.

A JMS destination is used to configure the properties of a JMS destination for the associated generic JMS provider. Connections to the JMS destination are created by the associated JMS connection factory. A JMS destination for use with a generic JMS provider (not the internal WebSphere JMS provider or MQSeries JMS provider) has the following properties.

Name:

The name by which the queue is known for administrative purposes within IBM WebSphere Application Server.

Data type	String

Description:

A description of the queue, for administrative purposes

JNDI Name:

The JNDI name of the actual (physical) name of the JMS destination bound into JNDI.

External JNDI Name:

The JNDI name that is used to bind the queue into the application server's name space.

As a convention, use the fully qualified JNDI name; for example, in the form jms/*Name*, where *Name* is the logical name of the resource.

This name is used to link the platform binding information. The binding associates the resources defined by the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Data type	String

Destination Type:

Whether this JMS destination is a queue (for point-to-point) or topic (for pub/sub).

Select one of the following options:

Queue
A JMS queue destination for point-to-point messaging.

Topic A JMS topic destination for pub/sub messaging.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Example: Configuring JMS Provider, JMS Connection Factory and JMS Destination settings for application clients

The purpose of this article is to help you to configure JMS Provider, JMS Connection Factory and JMS Destination settings.

- Required fields:
  - JMS Provider Properties page: name, and at least one protocol provider
  - JMS Connection Factory Properties page: name, jndiName, destination type
  - JMS Destination Properties page: name, jndiName, destination type
- Special cases:
  - The destination type must be `QUEUE`, or `TOPIC`.
- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_3" name="genericJMSProvider:name"
description="genericJMSProvider:description"
externalInitialContextFactory="genericJMSProvider:contextFactoryClass"
externalProviderURL="genericJMSProvider:providerUrl">
<classpath>genericJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms:GenericJMSDestination"
xmi:id="GenericJMSDestination_1" name="jmsDestination:name"
jndiName="jmsDestination:jndiName" description="jmsDestination:description"
externalJNDIName="jmsDestination:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_15">
<resourceProperties xmi:id="J2EEResourceProperty_17" name="jmsDestination:custonName"
value="jmsDestination:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms:GenericJMSConnectionFactory"
xmi:id="GenericJMSConnectionFactory_1" name="jmsCF:name" jndiName="jmsCF:jndiName"
description="jmsCF:description" userID="jmsCF:user" password="{xor}NTIsHBllMT4yOg=="
externalJNDIName="jmsCF:externalJndiName" type="QUEUE">
<propertySet xmi:id="J2EEResourcePropertySet_16">
<resourceProperties xmi:id="J2EEResourceProperty_18" name="jmsCF:customName"
value="jmsCF:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_17">
<resourceProperties xmi:id="J2EEResourceProperty_19"
name="genericJMSProvider:customName" value="genericJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Configuring new connection factories for application clients

During this task, you create a new JMS connection factory configuration for your application client.

1. Click the JMS provider for which you want to create a connection factory in the tree. Do one of the following:

- Configure a new JMS provider.
- Click an existing JMS provider.

2. Expand the JMS provider to view its **JMS Connection Factories** folder.
3. Click the folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS connection factory properties in the resulting property dialog.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Configuring new Java Message Service destinations for application clients

During this task, you create new Java Message Service (JMS) destination configuration for your application client.

1. Click the JMS provider in the tree for which you want to create a destination. Do one of the following:
   - Configure a new JMS provider.
   - Click an existing JMS provider.
2. Expand the JMS provider to view its **JMS Destinations** folder.
3. Click the folder. Do one of the following:
   - Right-click the folder and click **New Factory**.
   - Click **Edit > New** on the menu bar.
4. Configure the JMS destination properties in the resulting property dialog.
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Example: Configuring MQ Queue and Topic connection factories and destination factories for application clients

The purpose of this article is to help you configure MQ Queue connection factory, MQ Topic connection factory, MQ Queue destination factory, and MQ Topic destination factory settings.

- Required fields:
  - MQ Queue Connection Factory Properties page: name, jndiName, transport type
  - MQ Topic Connection Factory Properties page: name, jndiName, broker version
  - MQ Queue Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName, targetClient
  - MQ Topic Factory Properties page: name, jndiName, persistence, priority, expiry, baseQueueName, targetClient
- Special cases:
  - The transport type must be `CLIENT`, or `BINDINGS`.
  - The Broker Version must be `MA0C`, or `MQSI`.
  - The port must be a numerical value between -2417483648 and 2417483647.
  - The CCSID must be a numerical value between -2417483648 and 2417483647.
  - The persistence value must be `APPLICATION_DEFINED`, `QUEUE_DEFINED`, `PERSISTENT` or, `NONPERSISTENT`.
  - The priority must be `APPLICATION_DEFINED`, `QUEUE_DEFINED`, or `SPECIFIED`.
  - The expiry must be `APPLICATION_DEFINED`, `UNLIMITED`, or `SPECIFIED`.
  - The integer encoding must be `Normal`, or `Reversed`.

– The decimal encoding must be Normal, or Reversed.
– The floating encoding must be IEEENormal, IEEEReversed, S390.
– The target client must be JMS or MQ.
– On the MQ Queue Connection Factory Properites page, only set the queueManager, host, and portWhen (required) fields if the transport type is CLIENT.
– On the MQ Topic Connection Factory Properites page, only set the queueManager, host, and port (required) fields if the transport type is CLIENT.
– On the the MQ Topic Factory Properties, and the MQ Queue Factory Properties pages, only set the Integer encoding, decimal encoding, and floating point encoding (required) fields if you do not set nativeEncoding.
– On the MQ Topic Factory Properties, and the MQ Queue Factory Properties pages, the specified priority entry field must be an integer between 0 and 9 if priority is set to SPECIFIED .
– On the the MQ Topic Factory Properties, and the MQ Queue Factory Properties pages, the specified expiry entry field must be a value greater than 0 if expiry is set to SPECIFIED.

- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_1" name="MQ JMS Provider"
description="mqJMSProvider:description"
externalInitialContextFactory="mqJMSProvider:contextFactoryClass"
externalProviderURL="mqJMSProvider:providerUrl">
<classpath>mqJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms.mqseries:MQQueueConnectionFactory"
xmi:id="MQQueueConnectionFactory_1" name="mqQCF:name" jndiName="mqQCF:jndiName"
description="mqQCF:description" userID="mqQCF:user" password="{xor}Mi4OHBllMT4yOg=="
queueManager="mqQCF:queueManager" host="mqQCF:host" port="1" channel="mqQCF:channel"
transportType="CLIENT" clientID="mqQCF:clientId" CCSID="2">
<propertySet xmi:id="J2EEResourcePropertySet_3">
<resourceProperties xmi:id="J2EEResourceProperty_3" name="mqQCF:customName"
value="mqQCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.mqseries:MQTopicConnectionFactory"
xmi:id="MQTopicConnectionFactory_1" name="mqTCF:name" jndiName="mqTCF:jndiName"
description="mqTCF:description" userID="mqTCF:user"
password="{xor}Mi4LHBllNTE7NhE+Mjo=" host="mqTCF:host" port="1"
transportType="CLIENT" channel="mqTCF:channel" queueManager="mqTCF:queueManager"
brokerControlQueue="mqTCF:brokerControlQueue"
brokerQueueManager="mqTCF:brokerQueueManager" brokerPubQueue="mqTCF:brokerPubQueue"
brokerSubQueue="mqTCF:brokerSubQueue" brokerCCSubQ="mqTCF:brokerCCSubQ"
brokerVersion="MA0C" clientID="mqTCF:clientId" CCSID="2">
<propertySet xmi:id="J2EEResourcePropertySet_4">
<resourceProperties xmi:id="J2EEResourceProperty_4" name="mqTCF:customName"
value="mqTCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.mqseries:MQQueue" xmi:id="MQQueue_1" name="mqQ:name"
jndiName="mqQ:jndiName" description="mqQ:description" persistence="APPLICATION_DEFINED"
priority="SPECIFIED" specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1"
baseQueueName="mqQ:baseQueueName" baseQueueManagerName="mqQ:baseQueueManagerName"
CCSID="1" integerEncoding="Normal" decimalEncoding="Normal"
floatingPointEncoding="IEEENormal" targetClient="JMS">
<propertySet xmi:id="J2EEResourcePropertySet_5">
<resourceProperties xmi:id="J2EEResourceProperty_5" name="mqQ:customName"
value="mqQ:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.mqseries:MQTopic" xmi:id="MQTopic_1"
name="mqT:name" jndiName="mqT:jndiName" description="mqT:description"
persistence="APPLICATION_DEFINED" priority="SPECIFIED" specifiedPriority="1"
expiry="SPECIFIED" specifiedExpiry="2" baseTopicName="mqT:baseTopicName" CCSID="3"
integerEncoding="Normal" decimalEncoding="Normal" floatingPointEncoding="IEEENormal"
```

```
targetClient="JMS" brokerDurSubQueue="mqT:brokerDurSubQueue"
brokerCCDurSubQueue="mqT:brokerCCDurSubQueue">
<propertySet xmi:id="J2EEResourcePropertySet_6">
<resourceProperties xmi:id="J2EEResourceProperty_6" name="mqT:customName"
value="mqT:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_7">
<resourceProperties xmi:id="J2EEResourceProperty_7" name="mqJMSProvider:customName"
value="mqJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Example: Configuring WAS Queue and Topic connection factories and destination factories for application clients

The purpose of this article is to help you to configure WAS Queue connection factory, WAS Topic connection factory, WAS Queue destination factory, and WAS Topic destination factory settings.

- Required fields:
  - JMS Provider Properties page: name
  - WAS Queue Connection Factory Properties page: name, jndiName, node
  - WAS Topic Connection Factory Properties page: name, jndiName, node, port
  - WAS Queue Factory Properties page: name, jndiName, node, persistence, priority, expiry
  - WAS Topic Factory Properties page: name, jndiName, topic name, persistence, priority, expiry
- Special cases:
  - The port must be `QUEUED` or `DIRECT`.
  - The CCSID must be a numerical value between -2417483648 and 2417483647.
  - The persistence value must be `APPLICATION_DEFINED`, `PERSISTENT`, or `NONPERSISTENT`.
  - The priority must be `APPLICATION_DEFINED`, or `SPECIFIED`.
  - The expiry must be `APPLICATION_DEFINED`, `UNLIMITED`, or `SPECIFIED`.
  - On the WAS Topic Factory Properties, and the WAS Queue Factory Properties pages, the specified priority entry field must be an integer between 0 and 9 if priority is set to `SPECIFIED` .
  - On the WAS Topic Factory Properties, and the WAS Queue Factory Properties pages, the specified expiry entry field must be an value greater than 0 if expiry is set to `SPECIFIED`.
- Example:

```
<resources.jms:JMSProvider xmi:id="JMSProvider_2" name="WebSphere JMS Provider"
description="wasJMSProvider:description"
externalInitialContextFactory="wasJMSProvider:contextfactoryclass"
externalProviderURL="wasJMSProvider:providerUrl">
<classpath>wasJMSProvider:classpath</classpath>
<factories xmi:type="resources.jms.internalmessaging:WASQueueConnectionFactory"
xmi:id="WASQueueConnectionFactory_1" name="wasQCF:name" jndiName="wasQCF:jndiName"
description="wasQCF:description" userID="wasQCF:user" password="{xor}KD4sDhwZZSosOi0="
node="wasQCF:Node">
<propertySet xmi:id="J2EEResourcePropertySet_8">
<resourceProperties xmi:id="J2EEResourceProperty_8" name="wasQCF:customName"
value="wasQCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASTopicConnectionFactory"
xmi:id="WASTopicConnectionFactory_1" name="wasTCF:name" jndiName="wasTCF:jndiName"
description="wasTCF:description" userID="wasTCF:user" password="{xor}KD4sCxwZZTE+Mjo="
node="wasTCF:node" port="QUEUED" clientID="wasTCF:clientId">
<propertySet xmi:id="J2EEResourcePropertySet_9">
<resourceProperties xmi:id="J2EEResourceProperty_9" name="wasTCF:customName"
```

```
value="wasTCF:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASQueue" xmi:id="WASQueue_1"
name="wasQ:name" jndiName="wasQ:jndiName" description="wasQ:description"
node="wasQ:node" persistence="APPLICATION_DEFINED" priority="SPECIFIED"
specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1">
<propertySet xmi:id="J2EEResourcePropertySet_10">
<resourceProperties xmi:id="J2EEResourceProperty_10" name="wasQ:customName"
value="wasQ:customValue"/>
</propertySet>
</factories>
<factories xmi:type="resources.jms.internalmessaging:WASTopic" xmi:id="WASTopic_1"
name="wasT:name" jndiName="wasT:jndiName" description="wasT:description"
topic="wasT:topicName" persistence="APPLICATION_DEFINED" priority="SPECIFIED"
specifiedPriority="1" expiry="SPECIFIED" specifiedExpiry="1">
<propertySet xmi:id="J2EEResourcePropertySet_11">
<resourceProperties xmi:id="J2EEResourceProperty_11" name="wasT:customName"
value="wasT:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_12">
<resourceProperties xmi:id="J2EEResourceProperty_12" name="wasJMSProvider:customName"
value="wasJMSProvider:customValue"/>
</propertySet>
</resources.jms:JMSProvider>
```

Configuring new resource environment providers for application clients

During this task, you create new resource environment provider configurations for your application client.

To configure a new resource environment provider, perform the following steps:

1. Start the tool and open the EAR file for which you want to configure the new JMS provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file in which you want to configure the new JMS provider.
3. Expand the JAR file to view its contents.
4. Click the folder called **Resource Environment Providers**. Do one of the following:
   - Right-click the folder and click **New Provider**.
   - Click **Edit > New** on the menu bar.
5. Configure the JMS provider properties in the resulting property dialog.
6. Click **OK** when finished.
7. Click **File > Save** on the menu bar to save your changes.

Resource environment provider settings for application clients

Use this page to specify resource environment entry properties.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file. Right-click **Resource Environment Providers** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the administrative name for the resource environment provider.

Description:

Specifies a description of the resource environment provider for your administrative records.

Class Path:

Specifies the path to the JAR file that contains the implementation classes for the resource environment provider.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Configuring new resource environment entries for application clients

During this task, you create new resource environment entries for your client application.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file for which you want to configure the new resource environment entry. The EAR file contents are in the displayed tree view.
3. Click the desired resource environment provider, and complete the following action to configure new providers:
   - Configure a new resource environment provider.
4. Expand the resource environment provider to view the **resource environment entries** folder.
5. Click the folder. Complete one of the following actions:
   - Right-click the folder and select **New Factory**.
   - Click **Edit > New** on the menu bar.
6. Configure the data source properties in the resulting property dialog.
7. Click **OK**.
8. Click **File > Save** on the menu bar to save your changes.

Resource environment entry settings for application clients

Use this page to specify resource environment entry properties.
To view this Application Client Resource Configuration Tool (ACRCT) page, click **File** > **Open**. After you browse for an EAR file, click **Open**. Expand the selected JAR file > **Resource Environment Providers** > *resource environment instance*. Right-click **Resource environment entry** > and click **New**. The following fields appear on the **General** tab:

Name:

Specifies the administrative name for the resource environment entry.

Description:

Specifies a description of the URL for your administrative records.

JNDI Name:

Specifies the Java Naming and Directory Interface (JNDI) name for the resource, including any naming subcontexts.

Use this name to link to the binding information of the platform. The binding associates the resources defined in the deployment descriptor of the module to the actual (physical) resources bound into JNDI by the platform.

Custom Properties:

Specifies name-value pairs for setting additional properties on the object that is created at runtime for this resource.

You must enter a name that is a public property on the object and a value that can be converted from a string to the type required by the set method of the property. The acceptable properties and values depend on the object that is created. Refer to the object documentation for a list of valid properties and values.

Managing application clients

Perform the following tasks after deploying application clients.

After deploying application clients on z/OS or OS/390, you might want to or need to update the resources that you configured for those clients. To do so, you may complete one of the following tasks:
- Run the Application Client Resource Configuration Tool (ACRCT) on Windows, according to the steps below, and then reinstall the application on z/OS; or
- Run the ACRCT scripting tool on z/OS.

Note: This task only applies to J2EE application clients.
1. Update data source and data source provider configurations.
2. Update URLs and URL provider configurations.
3. Update mail session configurations.
4. Update JMS provider, connection factories, and destination configurations.
5. Update MQ JMS provider, MQ connection factories, and MQ destination configurations.
6. Update Resource Environment Entry and Resource Environment Provider configurations.
7. (Optional) Remove application client resources.

Updating data source and data source provider configurations with the Application Client Resource Configuration Tool

During this task, you update the configuration of an existing data source or data source provider.
1. Start the tool and open the EAR file containing the data source or data source provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the data source or data source provider to update.

3. Expand the JAR file to view its contents until you locate the particular data source or data source provider to update. Do one of the following:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. For detailed field help, go to:
   - Data source provider properties
   - Data source properties
5. Click **OK** when finished.
6. Click **File > Save** on the menu bar to save your changes.

Updating URLs and URL provider configurations for application clients

1. Start the tool and open the EAR file containing the URL or URL provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the URL or URL provider to update.
3. Expand the JAR file to view its contents.
4. Keep expanding the JAR file contents until you locate the particular URL or URL provider to update. Do one of the following:
   a. Right-click the object and click **Properties**
   b. Click **Edit** > **Properties** on the menu bar.
5. Update the properties in the resulting property dialog.
6. Click **OK** when finished.
7. Click **File** > **Save** to save your changes on the menu bar.

Updating mail session configurations for application clients

During this task, you update the configuration of an existing JavaMail session.

Note:

You cannot update the name of the default JavaMail provider. Also, you cannot delete the default JavaMail provider from the tree.

1. Start the tool and open the EAR file containing the JavaMail session. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the JavaMail session to update.
3. Expand the JAR file to view its contents.
4. Keep expanding the JAR file contents until you locate the particular JavaMail session to update. Do one of the following:
   a. Right-click the object and click **Properties**
   b. Click **Edit** > **Properties** from the menu bar.
5. Update the properties in the resulting property dialog.
6. Click **OK** when finished.
7. Select **File** > **Save** from the menu bar to save your changes.

Updating Jave Message Service provider, connection factories, and destination configurations for application clients

During this task, you update the configuration of an existing Java Message Service (JMS) provider, connection factory, or destination.

1. Start the tool and open the EAR file containing the JMS provider, connection factory, or destination. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the JMS provider, connection factory, or destination to update.
3. Expand the JAR file to view its contents until you locate the particular JMS provider, connection factory, or destination to update. When you find it, do one of the following:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. For detailed field help, see:
   - JMS provider properties
   - WAS Queue connection factory properties
   - WAS Topic connection factory properties
   - WAS Queue destination properties
   - WAS Topic destination properties
5. Click **OK**.
6. Click **File > Save** to save your changes.

Updating MQ Java Message Service provider, MQ connection factories, and MQ destination configurations for application clients

During this task, you will update the configuration of an existing MQ JMS provider, MQ connection factory, or MQ destination.

1. Start the Application Client Resource Configuration Tool (ACRCT).
2. Open the EAR file containing the MQ JMS provider, MQ connection factory, or MQ destination. The EAR file contents are displayed in a tree view.
3. Select the JAR file containing the MQ JMS provider, MQ connection factory, or MQ destination to update.
4. Expand the JAR file to view its contents until you locate the particular MQ JMS provider, MQ connection factory, or MQ destination that you want to update. Complete one of the following actions:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
5. Update the properties in the resulting property dialog. For detailed field help, see:
   - JMS provider properties
   - MQ Queue connection factory properties
   - MQ Topic connection factory properties
   - MQ Queue destination properties
   - MQ Topic destination properties
6. Click **OK**.
7. Click **File > Save** to save your changes.

Updating Resource Environment Entry and Resource Environment Provider configurations for application clients

During this task, you update the configuration of an existing Resource Environment Entry or Resource Environment Provider.

1. Start the tool and open the EAR file containing the Resource Environment Entry or Resource Environment Provider. The EAR file contents display in a tree view.
2. Select from the tree the JAR file containing the Resource Environment Entry or Resource Environment provider to update.
3. Expand the JAR file to view its contents until you locate the Resource Environment Entry or Resource Environment Provider to update. Do one of the following:
   - Right-click the object and click **Properties**.
   - Click **Edit > Properties** on the menu bar.
4. Update the properties in the resulting property dialog. For detailed field help, see:
   - Resource environment provider properties
   - Resource environment entry properties
5. Click **OK** when you finish.
6. Click **File > Save** on the menu bar to save your changes.

Example: Configuring Resource Environment settings

The purpose of this article is to help you configure Resource Environment settings.

- Required fields:
  - Resource Environment Provider page: name
  - Resource Environment Entry page: name, jndiName
- Example:

```
<resources.env:ResourceEnvironmentProvider xmi:id="ResourceEnvironmentProvider_1"
name="resourceEnvProvider:name" description="resourceEnvProvider:description">
<classpath>resourceEnvProvider:classpath</classpath>
<factories xmi:type="resources.env:ResourceEnvEntry" xmi:id="ResourceEnvEntry_1"
name="resourceEnvEntry:name" jndiName="resourceEnvEntry:jndiName"
description="resourceEnvEntry:description">
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
name="resourceEnvEntry:customName" value="resourceEnvEntry:customValue"/>
</propertySet>
</factories>
<propertySet xmi:id="J2EEResourcePropertySet_21">
<resourceProperties xmi:id="J2EEResourceProperty_23"
name="resourceEnvProvider:customName" value="resourceEnvProvider:customValue"/>
</propertySet>
</resources.env:ResourceEnvironmentProvider>
```

Example: Configuring Resource Environment custom settings for application clients

The purpose of this article is to help you configure Resource Environment custom settings.

- The custom page applies to every resource type. You can specify as many custom names and values as you need.
- Example:

```
<propertySet xmi:id="J2EEResourcePropertySet_20">
<resourceProperties xmi:id="J2EEResourceProperty_22"
    name="resourceEnvEntry:customName"
    value="resourceEnvEntry:customValue"/>
</propertySet>
```

Removing application client resources

Note: This task only applies to J2EE application clients.

To remove resources for application clients running on z/OS or OS/390, you may complete one of the following tasks:

- Run the Application Client Resource Configuration Tool (ACRCT) on Windows, according to the steps below, and then reinstall the application on z/OS; or
- Run the ACRCT scripting tool on z/OS.

1. Start the Application Client Resource Configuration Tool (ACRCT) and open the EAR file from which you want to remove an object. The EAR file contents display in a tree view. If you already have an EAR file open, and have made some changes, click **File** > **Save** to save your work before preceding to delete an object.
2. Locate the object that you want to remove in the tree.
3. Right-click the object, then click **Delete**.
4. Click **File** > **Save**.

The option to delete an item does not offer a confirmation dialog. As a safeguard, consider saving your work right before you begin this task. If you change your mind after removing an item, you can close the EAR file without saving your changes, canceling your deletion. Remember to close the EAR file immediately after the deletion, or you also lose any unsaved work that you performed since the deletion.

Running application clients

The J2EE specification requires support for a client container that runs stand-alone Java applications (known as J2EE application clients) and provides J2EE services to the applications. J2EE services include naming, security, and resource connections.

You are ready to run your application client using this tool after you have:

1. Written the application client program.
2. Assembled and installed an application module (`.ear` file) in the application server run time.
3. Deployed the application using the Application Client Resource Configuration Tool (ACRCT) on Windows, or the ACRCT scripting tool on z/OS.

Note: This task only applies to J2EE application clients.

1. On z/OS or OS/390, invoke the following script to launch J2EE application clients using the launchClient shell:

   ```
   install_root/bin/launchClient.sh
   ```

   Example invocation on z/OS:

   ```
   /usr/lpp/WebSphere/V5R0M0/bin/launchClient.bat
   ```

   The launchClient batch command:

   - Initializes the client run time.
   - Loads the class that you designated as the main class with the Application Assembly Tool (AAT) or the Assembly Toolkit.
   - Runs the main method of the application client program.

   When your program terminates, the application client run time cleans up the environment and the Java Virtual Machine code ends.

2. Pass parameters to the **launchClient** command. You can pass parameters to your application client program as well. The launchClient command allows you to do both. The launchClient command requires that the first parameter is either:
   - An EAR file specifying the application client to launch.
   - A request for launchClient usage information.

   All other parameters intended for the **launchClient** command must begin with the -CC prefix.

   Parameters that are not EAR files, or usage requests, or that do not begin with the -CC prefix, are ignored by the application client run time, and are passed directly to the application client program.

   The **launchClient** command retrieves parameters from three places:
   a. The command line
   b. A properties file
   c. System properties

   The parameters are resolved in the order listed above, with command line values having the highest priority and system properties the lowest. This prioritization allows you to set and override default values.
3. Specify the server name. By default, the **launchClient** command uses the environment variable `COMPUTERNAME` for the `BootstrapHost` property value. This setting is effective for testing your application client when it is installed on the same computer as the server. However, in other cases override this value with the name of your server.

   You can override the `BootstrapHost` value by invoking `launchClient` with the following parameters:

   ```
   launchClient myapp.ear -CCBootstrapHost=abc.midwest.mycompany.com
   ```

   You can also override the default by specifying the value in a properties file and passing the file name to the launchClient shell.

   Note: Security is controlled by the server. You do not need to configure security on the client because the client assumes that security is enabled. If security is not enabled, the server ignores the security request, and the application client works as expected.

You can store launchClient values in a properties file, a good method for distributing default values. You can then override one or more values on the command line. The format of the file is one launchClient `-CC` parameter per line without the `-CC` prefix. For example:

```
 verbose=true classpath=/usr/lpp/mydir/util.jar;/usr/lpp/mydir/harness.jar;/usr/lpp
/production/G19/global.jar BootstrapHost=abc.westcoast.mycompany.com tracefile=/usr
/lpp/WebSphere/mylog.txt
```

launchClient tool

This section describes the command line syntax for the Java TM2 Platform, WebSphere Application Server Enterprise (J2EE) launchClient tool.

The command line invocation syntax for the launchClient tool follows:

```
launchClient [<userapp.ear> |-help|-?] [-CCname=value] [app args]
```

where *userapp.ear* is the path and the name of the EAR file that contains the application client, *name* is the name of the parameter, *value* is the value to which the parameter ID is set, and *app args* are arguments that pass to the application client.

To print the usage information, the first parameter must be a path and a name to an EAR file, `-help`, or `-?`. All other parameters are optional and can appear in any order. The application client run time ignores any optional parameters that do not begin with a `-CC` prefix, and passes them to the application client.

Parameters

Supported arguments include:

-CCsoapConnectorPort
: The soap connector port. If you do not specify this argument, the WebSphere Application Server default value is used.

-CCverbose
: This option displays additional information messages. The default is `false`.

-CCclasspath
: A class path value. When you launch an application, the system class path is not used. If you want to access classes that are not in the EAR file or part of the resource class paths, specify the appropriate class path here. Multiple paths can be concatenated.

-CCjar
: The name of the client JAR file that resides within the EAR file for the application you wish to launch. Use this argument when you have multiple client JAR files in the EAR file.

-CCadminConnectorHost
: Specifies the host name of the server from which configuration information is retrieved. The default is the value of the `-CCBootstrapHost` parameter or the value of the local host if the `-CCBootstrapHost` parameter is not specified.

-CCadminConnectorPort
: Indicates the port number that the administrative client function should use. The default value is `8880` for SOAP connections and `2809` for RMI connections.

-CCadminConnectorType
: Specifies how the administrative client should connect to the server. Specify RMI to use the RMI connection type or specify SOAP to use the SOAP connection type. The default value is `SOAP`.

-CCadminConnectorUser
: Administrative clients use this user name when a server requires authentication. If the connection type is SOAP, and security is enabled on the server, this parameter is required. The SOAP connector does not prompt for authentication.

-CCadminConnectorPassword
: The password for the user name that the `-CCadminConnectorUser` parameter specifies.

-CCaltDD
: The name of an alternate deployment descriptor file. This parameter is used with the `-CCjar` parameter to specify the deployment descriptor to use. Use

this argument when a client jar file is configured with more than one deployment descriptor. Set the value to `null` to use the client JAR file standard deployment descriptor.

-CCBootstrapHost
The name of the host server you want to connect to initially. The format is: *`your.server.ofchoice.com`*

-CCBootstrapPort
The server port number. If you do not specify this argument, the WebSphere Application Server default value is `used`.

-CCproviderURL
Provides bootstrap server information that the initial context factory can use to obtain an initial context. WebSphere Application Server initial context factory can use either a CORBA object URL or an IIOP URL. CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. This value can contain more than one bootstrap server address. This feature can be used when attempting to obtain an initial context from a server cluster. You can specify bootstrap server addresses, for all servers in the cluster, in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. The address list does not process in a particular order. For naming operations, this value overrides the `-CCBootstrapHost` and `-CCBootstrapPort` parameters. An example of a CORBA object URL specifying multiple systems follows:

```
-CCproviderURL=corbaloc:iiop:myserver.mycompany.com:9810,:mybackupserver.mycompany.com:2809
```

This value is mapped to the `java.naming.provider.url` system property.

-CCinitonly
Use this option to initialize application client run time for ActiveX application clients without launching the client application. The default is `false`.

-CCtrace
Use this option to obtain debug trace information. You might need this information when reporting a problem to IBM Service. The default is `false`.

-CCtracefile
The name of the file to write trace information. The default is to output to the console.

-CCpropfile
Name of a properties file that contains launchClient properties. Specify the properties without the `-CC` prefix in the file. For example: `verbose=true`.

-CCsecurityManager
Enables and runs the WebSphere Application Server with a security manager. The default is `disable`.

-CCsecurityMgrClass
The fully qualified name of a class that implements a security manager. Only use this argument if the `-CCsecurityManager` parameter is set to enable. The default is `java.lang.SecurityManager`.

-CCsecurityMgrPolicy
The name of a security manager policy file. Only use this argument if the `-CCsecurityManager` parameter is set to enable. When you enable this parameter, the `java.security.policy` system property is set. The default is `<install_root>/ properties/client.policy`.

-CCD
Use this option to have the WebSphere Application Server set the specified system property during initialization. Do not use the = character after the -CCD. For example: -CCDcom.ibm.test.property=testvalue. You can specify multiple -CCD parameters. The general format of this parameter is -CCD<property key>=<property value>.

-CCexitVM
Use this option to have the WebSphere Application Server call System.exit() after the client application completes. The default is false.

-CCdumpJavaNameSpace
Prints out the Java portion of the WebSphere Application Server Java Naming and Directory Interface (JNDI) name space. The true value uses the short format which prints out the binding name and the type of the object bound at that location. The long value uses the long format which prints out the binding name, bound object type, local object, type, and string representation of the local object, for example: IORs, and string values. The default value is false.

-CCtraceMode
Specifies the trace format to use for tracing. If the valid value, basic, is not specified the default is advanced. Basic tracing format is a more compact form of tracing.

The following examples demonstrate correct syntax.
On the Windows operating system:

```
launchClient c:\earfiles\myapp.ear -CCBootstrapHost=myWASServer
-CCverbose=true app_parm1 app_parm2
```

Specifying the directory for an expanded EAR file

Each time launchClient is called, it extracts the EAR file to a random directory name in the temporary directory on your hard drive. Then it sets up the thread ClassLoader to use the extracted EAR file directory and JAR files included in the Manifest.mf client JAR file. In a normal J2EE Java client, these files are automatically cleaned up after the application exits. This cleanup occurs when the client container shutdown hook is called. To avoid extracting the EAR file (and removing the temporary directory) each time launchClient is called, complete the following steps:

1. Specify a directory to extract the EAR file by setting the com.ibm.websphere.client.applicationclient.archivedir Java system property. If the directory does not exist or is empty, the EAR file is extracted normally. If the EAR file was previously extracted, the launchClient tool reuses the directory.
2. Delete the directory before running the launchClient tool again, if you need to update your EAR file. When you call the launchClient command, it extracts the new EAR file to the directory. If you do not delete the directory or change the system property value to point to a different directory, launchClient reuses the currently extracted EAR file, and does not use your changed EAR file.

   Note: When specifying the com.ibm.websphere.client.applicationclient.archivedir property, make sure that the directory you specify is unique for each EAR file you use. For example, do not point MyEar1.ear and MyEar2.ear files to the same directory.

Application client troubleshooting tips

This section provides some debugging tips for resolving common J2EE application client problems. To use this troubleshooting guide, review the trace entries for one of the J2EE application client exceptions, and then locate the exception in the guide. Some of the errors in the guide are samples; the actual error you receive can be slightly different than what is shown here. Also, it can be useful to rerun the **launchClient** command specifying the `-CCverbose=true` option. This option provides additional information when the J2EE application client run time is initializing

Error: java.lang.NoClassDefFoundError

Explanation	This exception is thrown when Java code cannot load the specified class.
Possible causes	• Invalid or non-existent class • Classpath problem • Manifest problem

Recommended response

Check to determine if the specified class exists in a JAR file within your EAR file. If it does, make sure the path for the class is correct. For example, if you get the exception:

```
java.lang.NoClassDefFoundError:
WebSphereSamples.HelloEJB.HelloHome
```

ensure the class HelloHome exists in one of the JAR files in your EAR file. If it exists, ensure the path for the class is WebSphereSamples.HelloEJB.

If both the class and path are correct, then it is a classpath issue. Most likely, you do not have the failing class JAR file specified in the client JAR file manifest. To verify this situation, perform the following steps:

1. Open your EAR file with the Application Assembly Tool and click on the Application Client.
2. Add the names of the other JAR files in the EAR file to the Classpath field.

This exception is generally caused by a missing EJB module name from the Classpath field.

If you have multiple JAR files to enter in the Classpath field, be sure to separate the JAR names with spaces.

If you still have the problem, you have a situation where a class is loaded from the file system instead of the EAR file. This is a very difficult situation to debug because the offending class is not the one specified in the exception. Instead, another class is loaded from the file system before the one specified in the exception. To correct this problem, review the classpaths specified with the -CCclasspath option and the classpaths configured with the Application Client Resource Configuration Tool, or the Client Container Resource Configuration Scripting tool for z/OS. Look for classes that also exist in the EAR file. You must resolve the situation where one of the classes is found on the file system instead of in the `.ear` file. Remove entries from the classpaths, or include the `.jar` files and classes in the `.ear` file instead of referencing them from the file system.

If you use the -CCclasspath parameter or resource classpaths in the Application Client Resource Configuration Tool, or the Client Container Resource Configuration Scripting tool for z/OS, and you have configured multiple JAR files or classes, verify they are separated with the correct character for your operating system. Unlike the classpath field in the Application Assembly Tool, these classpath fields use platform-specific separator characters, usually a colon or a semi-colon (on Windows systems).

Note: The system classpath is not used by the Application Client run time if you use the launchClient batch or shell files. In this case, the system classpath would not cause this problem. However, if you load the launchClient class directly, you do have to search through the system classpath as well.

Error: com.ibm.websphere.naming.CannotInstantiateObjectException: Exception occurred while attempting to get an instance of the object for the specified reference object. [Root exception is javax.naming.NameNotFoundException: xxxxxxxxxx]

Explanation	This exception occurs when you perform a lookup on an object that is not installed on the host server. Your program can look up the name in the local client Java Naming and Directory Interface (JNDI) name space, but received a NameNotFoundException exception because it is not located on the host server. One typical example is looking up an enterprise bean that is not installed on the host server that you access. This exception might also occur if the JNDI name you configured in your Application Client module does not match the actual JNDI name of the resource on the host server.
Possible causes	• Incorrect host server invoked • Resource is not defined • Resource is not installed • Application server is not started • Invalid JNDI configuration
Recommended response	If you are accessing the wrong host server, run the **launchClient** command again with the -CCBootstrapHost parameter specifying the correct host server name. If you are accessing the correct host server, use the WebSphere **dumpnamespace** command line tool to see a listing of the host server JNDI name space. If you do not see the failing object name, the resource is either not installed on the host server or the appropriate application server is not started. If you determine the resource is already installed and started, your JNDI name in your client application does not match the global JNDI name on the host server. Use the Application Assembly Tool to compare the JNDI bindings value of the failing object name in the client application to the JNDI bindings value of the object in the host server application. They must match.

Error: javax.naming.ServiceUnavailableException: A communication failure occurred while attempting to obtain an initial context using the provider url: ″iiop://[invalidhostname]″. Make sure that the host and port information is correct and that the server identified by the provider URL is a running name server. If no port number is specified, the default port number 2809 is used. Other possible causes include the network environment or workstation network configuration. Root exception is org.omg.CORBA.INTERNAL: JORB0050E: In Profile.getIPAddress(), InetAddress.getByName[invalidhostname] threw an UnknownHostException. minor code: 4942F5B6 completed: Maybe

Explanation	This exception occurs when you specify an invalid host server name.

Possible causes	• Incorrect host server invoked • Invalid host server name
Recommended response	Run the **launchClient** command again and specify the correct name of your host server with the -CCBootstrapHost parameter.

Error: javax.naming.CommunicationException: Could not obtain an initial context due to a communication failure. Since no provider URL was specified, either the bootrap host and port of an existing ORB was used, or a new ORB instance was created and initialized with the default bootstrap host of ″localhost″ and the default bootstrap port of 2809. Make sure the ORB bootstrap host and port resolve to a running name server. Root exception is org.omg.CORBA.COMM_FAILURE: WRITE_ERROR_SEND_1 minor code: 49421050 completed: No

Explanation	This exception occurs when you run the **launchClient** command to a host server that does not have the Application Server started. You also receive this exception when you specify an invalid host server name. This situation might occur if you do not specify a host server name when you run launchClient. The default behavior is for launchClient to run to localhost, because WebSphere Application Server does not know the name of your host server. This default behavior only works when you are running the client on the same computer with WebSphere Application Server is installed.
Possible causes	• Incorrect host server invoked • Invalid host server name • Invalid reference to `localhost` • Application server is not started • Invalid bootstrap port
Recommended response	If you are not running to the correct host server, run the launchClient command again and specify the name of your host server with the -CCBootstrapHost parameter. Otherwise, start the Application Server on the host server and run the launchClient command again.

Error: javax.naming.NameNotFoundException: Name comp/env/ejb not found in context ″java:″

Explanation	This exception is thrown when the Java code cannot locate the specified name in the local JNDI name space.
Possible causes	• No binding information for the specified name • Binding information for the specified name is incorrect • Wrong class loader was used to load one of the program classes • A resource reference does not include any client configuration information

Recommended response	Open the EAR file with the Application Assembly Tool and check the bindings for the failing name. Ensure this information is correct. If you are using Resource References, open the EAR file with the Application Client Resource Configuration Tool, and make sure the Resource Reference has client configuration information and the name of the Resource Reference exactly matches the JNDI name of the client configuration. If it is correct, you might have a class loader issue.

Error: java.lang.ClassCastException: Unable to load class: org.omg.stub.WebSphereSamples.HelloEJB._HelloHome_Stub at com.ibm.rmi.javax.rmi.PortableRemoteObject .narrow(portableRemoteObject.java:269)

Explanation	This exception occurs when the application program attempts to narrow to the EJB home class and the class loaders cannot find the EJB client side bindings.
Possible causes	• The files, *_Stub.class and _Tie.class, are not in the EJB `.jar` file • Class loader could not find the classes
Recommended response	Look at the EJB `.jar` file located in the `.ear` file and verify the class contains the EJB client side bindings. These are class files whose names end in _Stub and _Tie. If these files are not present, then use the Application Assembly Tool to generate the binding classes. For more information, see article Generating deployment code for modules. If the binding classes are in the EJB `.jar` file, then you might have a class loader issue.

Error: WSCL0210E: The Enterprise archive file [EAR file name] could not be found. com.ibm.websphere.client.applicationclient.ClientContainerException: com.ibm.etools.archive.exception.OpenFailureException

Explanation	This error occurs when the application client run time cannot read the Enterprise Archive (EAR) file.
Possible causes	The most likely cause of this error is that the system cannot find the EAR file cannot be found in the path specified on the **launchClient** command.
Recommended response	Verify that the path and file name specified on the **launchclient** command are correct. If you are running on the Windows operating system and the path and file name are correct, use a short version of the path and file name (8 character file name and 3 character extension).

The launchClient command appears to hang and does not return to the command line when the client application has finished.

Explanation	When running your application client using the **launchClient** command the WebSphere Application Server run time might need to display the security login dialog. To display this dialog the WebSphere Application Server run time creates an Abstract Window Toolkit (AWT) thread. When your application returns from its main method to the application client run time, the application client run time attempts to return to the operating system and end the Java Virtual Machine code. However, since there is an AWT thread, the Java Virtual Machine code will not end until `System.exit` is called.
Possible causes	The Java Virtual Machine code does not end because there is an AWT thread. Java code requires that `System.exit()` be called to end AWT threads.
Recommended response	• Modify your application to call `System.exit(0)` as the last statement. • Use the `-CCexitVM=true` parameter when you call the **launchClient** command.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

Chapter 7. Using Web services based on Web Services for J2EE

Decide if a Web service implementation benefits your business process.

This topic introduces you to using Web services that are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification. WebSphere Application Server supports Web services that are developed and implemented based on Web Services for J2EE.

Use Web services when operating across a variety of platforms, including the J2EE 1.3 and non-J2EE platforms. Web services benefit your e-business solution by integrating these enterprise systems, especially systems that have developed over a long period of time.

Using Web services makes most sense if your application's clients are non-J2EE applications, unless you have J2EE applications spread across the Web. It is recommended that you use J2EE technologies if all your clients are J2EE applications because performance can decrease when you use a Web service in a J2EE exclusive environment.

Because Web services are easily applied to existing applications and information technology assets, new solutions can be deployed quickly and recomposed to address new opportunities. As Web services become more popular, the pool of services grows, promoting development of more robust models of just-in-time application and business integration over the Internet.

To use Web services applications with WebSphere Application Server:
1. Plan to use Web services. Review the Universal Description, Discovery, and Integration (UDDI), and Web Services Invocation Framework concepts to learn how these components can make your Web services plan more robust.
2. Migrate existing Web services.
3. Develop Web services.
4. Assemble Web services.
5. Deploy Web services.
6. Secure Web services.
7. Tune Web services.
8. Troubleshoot Web services.

The following is an example of how a business might use Web services.

The owner of a flower shop wants to start receiving orders from customers through the Web. She starts her venture by finding wholesale flower suppliers, pricing their product, and completing contracts for future flower orders.

Using Web services, the flower shop owner can find wholesale flower suppliers. One way she finds new suppliers is to use a UDDI registry to search for potential suppliers. She chooses the suppliers and the registry sends back information on how to contact the flower distributors that meet her criteria.

© Copyright IBM Corp. 2003

The flower shop owner can request price lists from each of the suppliers by obtaining a Web Services Description Language (WSDL) file for each potential supplier. The WSDL can be downloaded from the supplier's Web page, received through email, or retrieved from the supplier's UDDI registry entry.

The WSDL describes the procedure call. When using WebSphere Application Server, the procedure call is a Java API for XML-based remote procedure call (JAX-RPC), which helps her get price lists. The WSDL file also specifies the Universal Resource Locator (URL) where the request is to be sent.

The flower shop owner now has to compare the prices she received back from each supplier, decide which suppliers she is going to do business with, and make arrangements for future orders to be filled. The ground work has been laid for the flower shop to sell merchandise through the Web by using Web services to communicate with suppliers for the best prices and complete the ordering processes. The merchandise price lists need to be published to her Web site and she needs to provide a mechanism for customers to order flowers.

The flower supplier's Web services clients are deployed on the flower shop server. When a customer makes a transaction to purchase flowers through the Web, the order is sent to the supplier through JAX-RPC. The supplier responds by sending a confirmation with the order number and shipping date. The suppliers maintain the inventory and the flower shop owner handles billing and customer order management.

Similarly, the flower shop catalog can be composed automatically from the catalogs of all the suppliers. If the supplier ships directly to the customer, the order tracking inquiries can pass directly to the supplier's order tracking system. Web services can also be used by the supplier to send invoices for orders and by the flower shop to pay the supplier's invoices. Processes that previously required forms to be filled out manually, and faxed or mailed, can now be done automatically, saving labor costs for both the flower shop and the supplier.

Using Web services is beneficial because a much larger inventory is made available to the flower shop. There is no merchandise maintenance overhead, but the flower shop can offer their customers products that they otherwise might not have. Selling flowers through the Web increases capital for the flower shop without overhead of another store or money invested into additional product.

For a more detailed scenario, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporated the Web services concept.

Web services

Web services are self-contained, modular applications that you can describe, publish, locate, and invoke over a network.

WebSphere Application Server supports Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

A typical Web services scenario is a business application requesting a service from a given URL using SOAP messages over a Hypertext Transport Protocol (HTTP) or Java Messaging Service (JMS) transport. The service receives the request, processes it, and returns a response. Examples of a simple Web service include weather

reports and stock quotes. The method call is synchronous, that is, it waits until the result is available. Transaction Web services, supporting quotes, business-to-business (B2B) or business-to-client (B2C) operations include airline reservations or purchase orders.

Web services are Web applications that allow you to be more flexible in your business processes by integrating with applications that otherwise would not communicate. The inner-library loan program at your local library is a good example of the Web services concept and its evolution. The Web service concept existed even before the term; the concept exploded with the birth of the Internet. Before, you would visit your library, search the collections and check out your books. If you didn't find the book you wanted, the librarian did a search for you by computer or phone and located the book at nearby library. The librarian ordered the book for you and you picked it up after it was delivered to your local library. By incorporating Web services applications, you can streamline your library visit. Now, you can search the local library collection and other local libraries at the same time. When other libraries provide your library with a Web service to search their collection (the service could have been provided through UDDI), your results yield their resources as well. Another Web service application might enable you to check the book out and get it sent to your home. Using Web services applications saved you time and created a convenience for you, as well as freeing the librarian to do other business tasks. For a more detailed scenario, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporated the Web services concept.

A Web service can be the service itself or the client that accesses the service.

Web services reflect a new, service-oriented architecture approach to programming. This approach is based on the idea of building applications by discovering and implementing network-available services, or by invoking available applications to accomplish some task. Web services deliver interoperability, for example, the ability for components created in different programming languages to work together as if they were created using the same language. Web services rely on existing transport technologies, such as HTTP, and standard data encoding techniques, such as Extensible Markup Language (XML), for invoking the implementation.

The key components of a Web service are:

- Web Services Description Language (WSDL)

  WSDL is the XML-based file that describes the Web service and allows the Web service request to bind to the service.
- SOAP

  SOAP is the XML-based protocol that allows the Web service request to invoke the service.
- Universal Description, Discovery and Integration Protocol (UDDI)

  UDDI is the registry that hosts the service broker. UDDI is similar to the Yellow Pages in a phone book.

SOAP

SOAP is a specification for exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three key actors in a service oriented architecture (SOA): service provider, service requestor and service broker. Then main goal of its design is to be simple and extensible. A SOAP message is used to request a Web service.

SOAP was submitted to the World Wide Web Consortium (W3C) as the basis of the eXtensible Markup Language (XML) Protocol Working Group by several companies, including IBM and Lotus.

SOAP is an XML-based protocol that consists of three parts: an *envelope* that defines a framework for describing message content and process instructions, a set of *encoding rules* for expressing instances of application-defined data types, and a *convention* for representing remote procedure calls and responses.

SOAP is transport protocol-independent and can be used in combination with a variety of protocols. In Web services that are developed and implemented for use with WebSphere Application Server, SOAP is used in combination with HyperText Transport Protocol (HTTP), HTTP extension framework, and Java Messaging Service (JMS). SOAP is also operating system independent and not tied to any programming language or component technology.

Due to these characteristics, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages. Also, both server and client sides can reside on any suitable platform.

For more information about SOAP, see Web services: Resources for learning.

Planning to use Web services based on Web Services for J2EE

This topic discusses how to plan your use of Web services that are developed and implemented based on the Web Services for Java 2, Enterprise Edition (J2EE) specification.

Read the Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporated the Web services concept.

To plan to use Web services based on Web Services for J2EE:

1. Design Web services to fit your e-business solution. Consider what you want to accomplish by using Web services, how Web services fit into your current topology, applications and programming model. Decide how the Web services process requests on the server and how the clients manage and use the Web service.

   Design your Web services for reliability, availability, manageability and security. For example, you want your Web services to process a transaction in a reasonable time at all hours of the day and provide users with good security characteristics, such as authentication for buyers. Planning to use Web services to work with WebSphere Application Server helps to meet these requirements.

   To support Web services, extend WebSphere Application Server to support Web services standards. For interoperable Web services running on platforms supplied by multiple vendors, standards are essential.
2. Decide what development and implementation tools to use. You can use a variety of manual development and implementation tasks. Whether you have an existing Web service to implement or you want to develop your own from a Java bean or enterprise JavaBean (EJB), you can choose different tasks respective to your resources. You can also use the the WebSphere Application Developer Studio to complete development and implementation tasks.

See Developing Web services based on Web Services for J2EE for information about developing Web services based on the Java language through the WebSphere Application Server. To read more about the WebSphere Application Developer Studio see the topic Web services development in the WebSphere Application Server Developer Studio InfoCenter.
3. Install WebSphere Application Server.
4. Review Web services Samples.

Develop a Web service.

Service-oriented architecture

A *Service-oriented architecture (SOA)* is a collection of services that communicate with each other, for example passing data from one service to another or coordinating an activity between one or more services.

Companies have longed to integrate existing systems in order to implement Information Technology (IT) support for business processes that cover the entire business value chain. A variety of designs are used, ranging from rigid point-to-point electronic data interchange (EDI) interactions to Web auctions. By using the Internet, companies make their IT systems available to internal departments or external customers, but the interactions are not flexible and are without standardized architecture.

Because of this increasing demand for technologies that support connecting and sharing of resources and data, there is a need for a flexible, standardized architecture. A service-oriented architecture (SOA) is a flexible architecture that unifies business processes by structuring large applications into building blocks, or small modular functional units or services, to be used by different groups of people in and outside the company. The building blocks can be one of three roles: service provider, service broker, or service requestor. See Web services approach to a service-oriented architecture to learn more about these roles.

Requirements for a SOA

In order to efficiently use a SOA, you must abide by the following requirements:

- **Interoperability between different systems and programming languages** .

  The most important basis for a simple integration between applications on different platforms is a communication protocol, which is available for most systems and programming languages.
- **Clear and unambiguous description language**.

  To use a service offered by a provider, it is not only necessary to be able to access the provider system, but the syntax of the service interface must also be clearly defined in a platform-independent fashion.
- **Retrieval of the service**.

  To allow a convenient integration at design time or even system run time, a search mechanism is required to retrieve suitable services. The services should be classified as computer-accessible, hierarchical or taxonomies based on what the services in each category do and how they can be invoked.

Web services approach to a service-oriented architecture

Web services are a new technology that implement the service-oriented architecture (SOA). A major focus of Web services is to make functional building blocks accessible over standard Internet protocols that are independent from platforms and programming languages. These services can be new applications or just

wrapped around existing legacy systems to make them network-enabled. A service can rely on another service to achieve its goals.

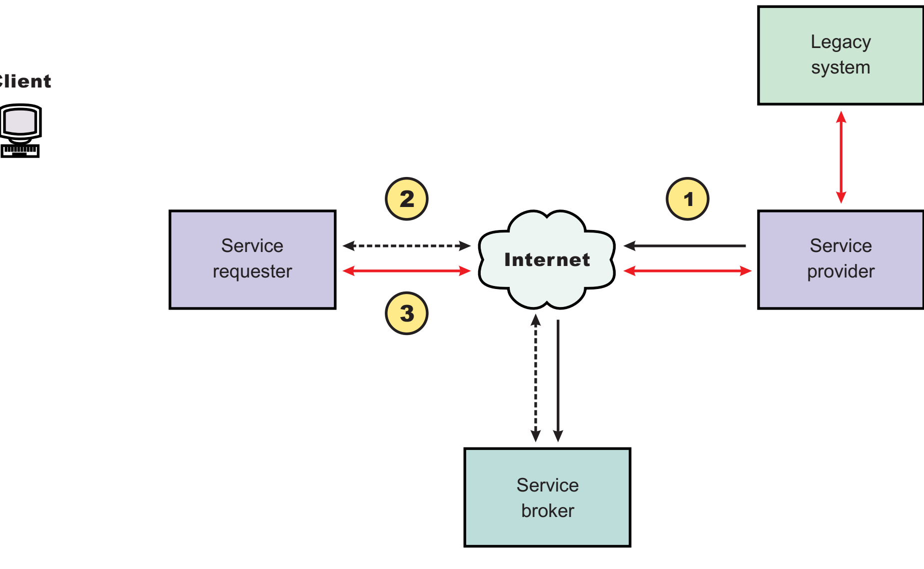
Each SOA building block can play one or more of three roles:

- **Service provider**

  The service provider creates a Web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or, if they are free, how to exploit them for other value. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service.
- **Service broker**

  The service broker, also known as service registry, is responsible for making the Web service interface and implementation access information available to any potential service requestor. The implementer of the broker have to decide about the scope of the broker. Public brokers are available all over the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, the width and breadth of the offered information has to be decided. Some brokers will specialize in breadth of listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. There are also brokers that catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, number of listings or accuracy of the listings.
- **Service provider**

  The service requestor locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.



.

Characteristics of the Web service architecture

The presented SOA employs a loose coupling between the participants, which provides greater flexibility in the following ways:

- A client is not coupled to a server, but to a service. Therefore, the integration of the server takes place outside the scope of the client application programs.
- Old and new functional blocks, or applications and systems, are encapsulated into components that work as services.
- Functional components and their interfaces are separated allowing new interfaces to be plugged in more easily.
- Within complex applications, the control of business processes can be isolated. A business rule engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.
- Services can be incorporated dynamically during run time.
- Bindings are specified using configuration files and can be easily adapted to new needs.

Properties of a service-oriented architecture

The service-oriented architecture offers the following properties:

- **Web services are self-contained.**

  On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, merely a Web server and a SOAP server are required. It is possible to Web services enable an existing application without writing a single line of code.
- **Web services are self-describing.**

  Neither the client nor the server knows or cares about anything besides the format and content of request and response messages (loosely coupled application integration). The definition of the message format travels with the message; no external metadata repositories or code generation tool are required.
- **Web services can be published, located, and invoked across the Internet.**

  This technology uses established lightweight Internet standards such as HTTP. It leverages the existing infrastructure. Some additional standards that are required to do so include SOAP, WSDL, and UDDI.
- **Web services are language-independent and interoperable.**

  Client and server can be implemented in different environments. Existing code does not have to be changed in order to be Web service enabled. In this redbook, however, we assume that Java is the implementation language for both the client and the server side of the Web service.
- **Web services are inherently open and standard-based.**

  XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open-source projects. Therefore, vendor independence and interoperability are realistic goals this time.
- **Web services are dynamic.**

  Dynamic e-business can become reality using Web services because, with UDDI and WSDL, the Web service description and discovery can be automated.
- **Web services are composable.**

  Simple Web services can be aggregated to more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This shortens development time and enables best-of-breed implementations.
- **Web services build on proven mature technology.**

There are a lot of commonalities, as well as a few fundamental differences to other distributed computing frameworks. For example, the transport protocol is text based and not binary.

- **Web services are loosely coupled.**

  Traditionally, application design has depended on tight interconnections at both ends. Web services require a simpler level of coordination that allows a more flexible re-configuration for an integration of the services in question.
- **Web services provide programmatic access.**

  The approach provides no graphical user interface; it operates at the code level. Service consumers need to know the interfaces to Web services but do not need to know the implementation details of services.
- **Web services provide the ability to wrap existing applications.**

  Already existing stand-alone applications can easily be integrated into the service-oriented architecture by implementing a Web service as an interface.

Web services business models supported

The properties and benefits of using a service-oriented architecture (SOA) such as Web services is well suited for binding small modules that perform independent tasks within a highly heterogeneous e-business model. Web services can be easily wrapped around existing applications in your business model and plugged into different business processes.

For connecting to a large monolithic system that does not allow the implementation of different flexible business processes, other approaches might be better suited, for example, to satisfy specialized features, such as performance or security.

The following business models are easily implemented by using an architecture including Web services:

- **Business information**

  Sharing of information with consumers or other businesses. Web services can be used to expand the reach through such services as news streams, local weather reports, integrated travel planning, intelligent agents, and so forth.
- **Business integration**

  Providing transactional, fee-based services for customers. A global value network of suppliers can be easily created. Web services can be implemented in auctions, e-marketplaces, reservation systems, and so forth.
- **Business process externalization**

  Web services can be used to model value chains by dynamically integrating processes to a new solution within an organizational unit or even with those of other e-businesses. This can be achieved by dynamically linking internal applications to new partners and suppliers, to offer their services to complement internal services.

To see how these models are implemented using all aspects of Web services, see Web services scenario: Overview which tells the story of a fictional online garden supply retailer named Plants by WebSphere and how they incorporate the Web services concept.

Migrating Apache SOAP Web services to Web Services for J2EE

If you have used Web services based on Apache SOAP in WebSphere Application Server Version 4.0.x through Version 5.0.2, and now want to develop and implement Web services based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification, you need to migrate your Version 4.0 and 5.0 client applications.

To migrate these client applications according to the Web Services for J2EE standards:

1. Plan your migration strategy. There are two ways you can port an Apache SOAP client to Java API for XML-based RPC (JAX-RPC) Web services client:
   - If you have, or can create, a Web Services Description Language (WSDL) document for the service, consider using the **WSDL2Java** command tool to generate bindings for the Web service. It is more work to adapt an Apache SOAP client to use the generated JAX-RPC bindings, but the resulting client code is more robust and easier to maintain. To follow this path, see Develop a Web services client based on Web Services for J2EE.
   - If you do not have a WSDL document for the service, do not expect the service to change, and you want to port the Apache SOAP client with a minimal work, you can convert the code to use the JAX-RPC dynamic invocation interface (DII), which is similar to the Apache SOAP APIs. The DII APIs do not use WSDL or generated bindings.

   You should be aware that since JAX-RPC does not specify a framework for user-written serializers, the JAX-RPC does not support the use of custom serializers. If your application cannot conform to the default mapping between Java, WSDL, and XML supported by WebSphere Application Server, you should not attempt to migrate the application. The remainder of this topic assumes that you have decided to use the JAX-RPC DII APIs.
2. Review the GetQuote sample. There is a Web services migration sample in the Samples Gallery. This sample is located in the `GetQuote.java` file, originally written for Apache SOAP, and includes an explanation about the changes needed to migrate to the JAX-RPC DII interfaces.
3. Convert the client application from Apache SOAP to JAX-RPC DII The Apache SOAP API and JAX-RPC DII API structures are similar. You can instantiate and configure a call object, set up the parameters, invoke the operation, and process the result in both. You can create a generic instance of a Service object with

   ```
   javax.xml.rpc.Service service =
    ServiceFactory.newInstance().createService(new QName(""));
   ```

   in JAX-RPC.
   a. Create the call object. An instance of the call object is created by

      ```
      org.apache.soap.rpc.Call call = new org.apache.soap.rpc.Call ()
      ```

      in Apache SOAP.

      An instance of the call object is created by

      ```
      java.xml.rpc.Call call = service.createCall();
      ```

      in JAX-RPC.
   b. Set the endpoint URI. The target URI for the operation is passed as a parameter to

      ```
      call.invoke:  call.invoke("http://...", "");
      ```

in Apache SOAP.

The setTargetEndpointAddress method is used as a parameter to

```
call.setTargetEndpointAddress("http://...");
```

in JAX-RPC.

Apache SOAP has a setTargetObjectURI method on the call object that contains routing information for the request. JAX-RPC has no equivalent method. The information in the targetObjectURI is included in the targetEndpoint URI for JAX-RPC.

c. Set the operation name. The operation name is configured on the call object by

```
call.setMethodName("opName");
```

in Apache SOAP.

The setOperationName method, which accepts a `QName` instead of a `String` parameter, is used in JAX-RPC as follows:

```
call.setOperationName(new javax.xml.namespace.Qname("namespace", "opName"));
```

d. Set the encoding style. The encoding style is configured on the call object by

```
call.setEncodingStyleURI(org.apache.soap.Constants.NS_URI_SOAP_ENC);
```

in Apache SOAP.

The encoding style is set by a property of the call object

```
call.setProperty(javax.xml.rpc.Call.ENCODINGSTYLE_URI_PROPERTY,
 "http://schemas.xmlsoap.org/soap/encoding/");
```

in JAX-RPC.

e. Declare the parameters and set the parameter values. Apache SOAP parameter types and values are described by parameter instances, which are collected into a Vector and set on the call object before the call, for example:

```
Vector params = new Vector ();
params.addElement (new org.apache.soap.rpc.Parameter(name, type,
 value, encodingURI));
// repeat for additional parameters...
call.setParams (params);
```

For JAX-RPC, the call object is configured with parameter names and types without providing their values, for example:

```
call.addParameter(name, xmlType, mode);
// repeat for additional parameters
call.setReturnType(type);
```

Where

- *name* (type `java.lang.String`) is the name of the parameter
- *xmlType* (type `javax.xml.namespace.QName`) is the XML type of the parameter
- *mode* (type `javax.xml.rpc.ParameterMode`) the mode of the parameter, for example, IN, OUT, or INOUT

f. Make the call. The operation is invoked on the call object by

```
org.apache.soap.Response resp = call.invoke(endpointURI, "");
```

in Apache SOAP.

The parameter values are collected into an array and passed to `call.invoke` as follows:

```
Object resp = call.invoke(new Object[] {parm1, parm2,...});
```

in JAX-RPC.

g. Check for faults. You can check for a SOAP fault on the invocation by checking the response:

```
if resp.generatedFault then {
org.apache.soap.Fault f = resp.getFault;
f.getFaultCode();
f.getFaultString();
}
```

in Apache SOAP.

A `java.rmi.RemoteException` is thrown in JAX-RPC if a SOAP fault occurs on the invocation.

```
try {
... call.invoke(...)
} catch (java.rmi.RemoteException) ...
```

h. Retrieve the result. In Apache SOAP, if the invocation was successful and returns a result, it can be retrieved from the Response object:

```
Parameter result = resp.getReturnValue(); return result.getValue();
```

In JAX-RPC, the result of invoke is the returned object when no exception is thrown:

```
Object result = call.invoke(...);
 ...
return result;
```

Developing a Web services client based on Web Services for J2EE.

Test the Web services-enabled clients.

Developing Web services based on Web Services for J2EE

This topic explains how to develop a Web service using the Web Services for Java 2, Enterprise Edition (J2EE) specification. Web services are structured in a service-oriented architecture (SOA) that makes integrating your business and e-commerce systems more flexible.

For more information about when and how you should to use Web services see Using Web services based on Web Services for J2EE. You can read about several concepts, including what is Web services, SOAP, WSDL, Web Services for J2EE and Java API for XML-based remote procedure call (JAX-RPC). If you would like to review a scenario where Web services are used, see Web services scenario: Overview.

WebSphere Application Server uses Web services standards developed for the Java language under the Java Community Process (JCP). These standards include the Web Services for J2EE and JAX-RPC specifications.

You can also use the WebSphere Studio Application Developer Version 5.1 graphical user interface development tools to develop Web services that integrate with WebSphere Application Server.

Before you develop a Web service you need to Set up a Web services development and unmanaged client execution environment and Install the Web Services Development Kit for z/OS.

Follow the Example: Developing Web services based on Web Services for J2EE for a step-by-step look at this task.

You can develop a Web service based on Web Services for J2EE in one of four ways:

1. Develop a Web service using a Java bean.
2. Develop a Web service using a stateless session enterprise bean.
3. Develop a Web service with an existing WSDL file using a Java bean.
4. Develop a Web service with an existing WSDL file using a stateless session enterprise bean.

Assemble the Web service.

Example: Developing Web services based on Web Services for J2EE

This example takes you through the steps to develop a Web service from an enterprise JavaBean (EJB) implementation. The development process is based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

1. **Select the EJB or Java bean implementation that you want to enable as a Web service.**

   The implementation must meet the following Web Services for J2EE specification requirements:

   - It must have methods that can be mapped to a Service Endpoint Interface. See step 2 for more information.
   - It must be a stateless session EJB or a Java bean without client-specific state, since the implementation bean might be selected to process a request from any client. If a client-specific state is required, a client identifier must be passed as a parameter of the Web service operation.

     The selected methods of an EJB must not have a transaction attribute of Mandatory, because there is no standard for Web services transactions at this time.

     A Java bean in a Web container requires the following:

     – A public default constructor
     – Exposed public methods
     – It must not save a client-specific state between method calls
     – It must be a public, non-final, and non-abstract class
     – It must not define a `finalize()` method
2. **Develop a Service Endpoint Interface**.

   Developing a Web service requires a Service Endpoint Interface.

   If you are using an EJB implementation, develop a Service Endpoint Interface from an EJB remote interface.

   If you are using a Java bean implementation, develop a Service Endpoint Interface for a Java bean implementation.
3. Develop a WSDL file.
4. **Develop deployment descriptor templates**.

If you are using an EJB implementation, develop Web services deployment descriptor templates from an EJB implementation.

If you are using a Java bean implementation, develop Web services deployment descriptor templates for a Java bean implementation.

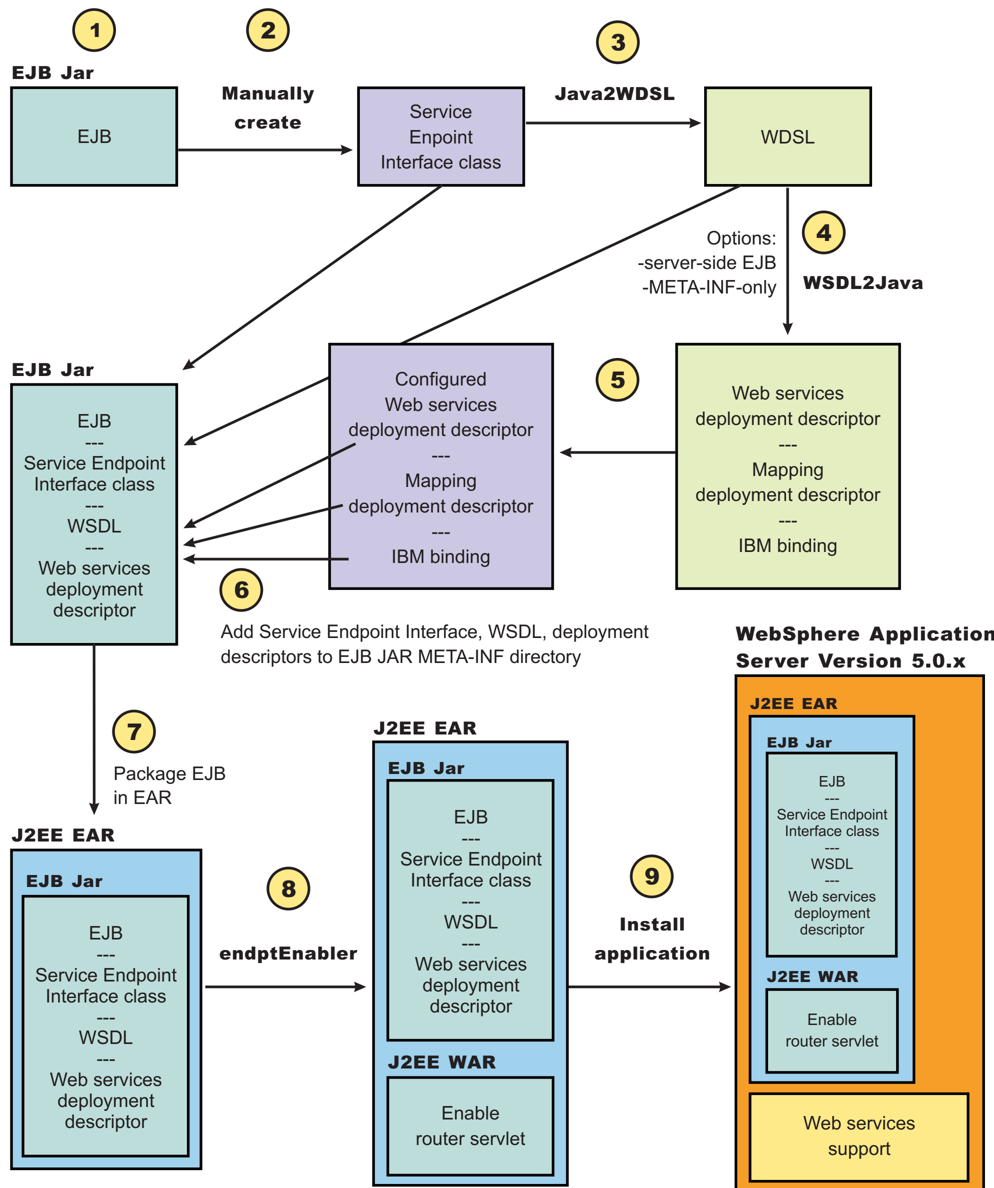
5. **Configure the deployment descriptors.**

   By setting the `ejb-link` or `servlet-link` values of the `service-impl-bean` elements you can link to the EJB or Java bean that implements the service.

   Configure the webservices.xml deployment descriptor.

   Configure the ibm-webservices-bnd.xmi deployment descriptor.
6. Assemble a JAR file or Assemble a WAR file.
7. Assemble an EAR file from a Jar file or Assemble an EAR file from a WAR file.
8. Enable the Web service-enabled EAR file.

   This step only applies if you are using an EJB implementation.
9. Deploy the Web service application.
10. Publish the WSDL file.



Web Services for J2EE

The *Web services for Java 2 platform, Enterprise Edition (J2EE)* specification defines the programming model and run-time architecture for implementing Web services based on the Java language. Another name for Web Services for J2EE is the Java Specification Requirements (JSR) 109. The specification includes open standards for developing and implementing Web services.

WebSphere Application Server Versions 5.0.2 and later use Web Services for J2EE as the standard for developing and implementing Web services. Before Version 5.0.2, WebSphere Application Server developed and implemented Web services based on Apache SOAP.

Web Services for J2EE focuses on eXtensible Markup Language (XML) remote procedure call (RPC) and the Java language, including representing XML-based interface definitions in the Java language; Java language definitions in XML-based definition languages, such as SOAP, and assembling.

Web Services for J2EE is the preferred platform for Web-based programming because it provides open standards allowing different types of languages, operating systems and software to communicate seamlessly through the Internet.

In order to achieve the benefits of using Web Services for J2EE, the Web services that you want to communicate with (provided by other sources), must also be based on the Java language. These other Web services can use other operating systems and languages, but the Web service itself must be based on the Java language.

For a Java application to act as Web service client, a mapping between the Web Services Description Language (WSDL) file and the Java application must exist. The mapping is defined by the Java API for XML-based RPC (JAX-RPC) specification. You can use a Java component to implement a Web service by specifying the component's interface and binding information in the WSDL file and designing the application server infrastructure to accept the service request. This entire process encompassed is based on the Web Services for J2EE specification.

Using Web Services for J2EE in WebSphere Application Server is based on J2EE 1.3. The same standards are included in J2EE 1.4.

To review the entire Web Services for J2EE specification, see Web services: Resources for learning.

Java API for XML-based remote procedure call (JAX-RPC)

The *Java API for XML-based RPC (JAX-RPC)* specification enables Java language developers to develop SOAP-based interoperable and portable Web services. JAX-RPC provides core APIs for developing and deploying Web services on a Java platform and is a required part of the J2EE 1.4 platform. JAX-RPC Web services can also be developed and deployed on J2EE 1.3 containers.

The JAX-RPC standard covers the programming model and bindings for using Web Services Description Language (WSDL) for Web services in the Java language.

JAX-RPC defines the mappings between the WSDL port types and the Java interfaces, as well as between Java language and eXtensible Markup Language (XML) schema types.

To learn more about JAX-RPC see Web services: Resources for learning.

Artifacts used to develop Web services based on Web Services for J2EE

Development artifacts enable an enterprise bean or a Java bean module to be a Web service. This topic describes artifacts used to develop Web services that are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

To create a Web service from an enterprise bean or a Java bean module, the following files are added to the respective Java archive (JAR) or Web archive (WAR) modules at assembly time:

- **Web Services Description Language (WSDL) eXtensible Markup Language (XML) file**

  The WSDL XML file describes the Web service being implemented.
- **Service Endpoint Interface**

A Service Endpoint Interface is the Java interface corresponding to the Web service port type implemented. The Service Endpoint Interface is defined by the WSDL 1.1 W3C Note.

- **webservices.xml**

  The `webservices.xml` file contains the J2EE Web service deployment descriptor specifying how the Web service is implemented. The `webservices.xml` file is defined in the Web Services for J2EE specification available through Web services: Resources for learning
- **ibm-webservices-bnd.xmi**

  This file contains WebSphere product-specific deployment information and is defined in ibm-webservices-bnd.xmi assembly properties.
- **Java API for XML-based remote procedure call (JAX-RPC) mapping file**

  The JAX-RPC mapping deployment descriptor specifies how Java elements are mapped to and from WSDL file elements.

The following files are added to an application client, enterprise JavaBean (EJB), or Web module to permit J2EE client access to Web services:

- **WSDL file**

  The WSDL file is provided by the Web service implementer.
- **Java interfaces for the Web service**

  The Java interfaces are generated from the WSDL file as specified by the JAX-RPC specification. These bindings are the Service Endpoint Interface based on the WSDL port type, or the service interface, which is based on the WSDL service.
- **webservicesclient.xml**

  The `webservicesclient.xml` file is the client-side deployment descriptor describing the services being accessed. The `webservicesclient.xml` file is defined in the Web Services for J2EE specification, available through Web services: Resources for learning.
- **ibm-webservicesclient-bnd.xmi**

  This file contains WebSphere product-specific deployment information such as security information.
- **Other JAX-RPC binding files**

  Additional JAX-RPC binding files that support the client application in mapping SOAP to Java language are generated from WSDL by the **WSDL2Java** command tool.

Mapping between Java language, WSDL and XML

This topic contains the mappings between the Java language and eXtensible Markup Language (XML) technologies, including XML Schema, Web Services Description Language (WSDL) and SOAP, supported by WebSphere Application Server. Most of these mappings are specified by the Java API for XML-based remote procedure call (JAX-RPC) specification. Some mappings that are optional or unspecified in JAX-RPC are also supported.

There are references to the JAX-RPC specification through this topic. You can review the JAX-RPC specification through Web services: Resources for learning.

The IBM Web Services Developer Kit for z/OS contains the **Java2WSDL** and **WSDL2Java** command-line tools needed for developing and implementing Web services. See Installing the IBM Web Services Developer Kit for z/OS to start using the tool.

Notational conventions

The following table specifies the namespace prefixes and corresponding namespaces used.

Namespace prefix	Namespace
xsd	http://www.w3.org/2001/XMLSchema
xsi	http://www.w3.org/2001/XMLSchema-instance
soapenc	http://schemas.xmlsoap.org/soap/encoding/
wsdl	http://schemas.xmlsoap.org/wsdl/
wsdlsoap	http://schemas.xmlsoap.org/wsdl/soap/
ns	user defined namespace
apache	http://xml.apache.org/xml-soap
wasws	http://websphere.ibm.com/webservices/

Detailed mapping information

The following sections identify the supported mappings, including:
- Java-to-WSDL mapping
- WSDL-to-Java mapping
- Mapping between WSDL and SOAP messages

Java-to-WSDL mapping

This section summarizes the Java-to-WSDL mapping rules. The Java-to-WSDL mapping rules are used by the **Java2WSDL** command tool for bottom-up processing. In bottom-up processing, an existing Java service implementation is used to create a WSDL file defining the Web service. The generated WSDL file can require additional manual editing for the following reasons:
- Not all Java classes and constructs have mappings to WSDL. For example, Java classes that do not comply with the Java bean specification rules might not map to a WSDL construct.
- Some Java classes and constructs have multiple mappings to WSDL. For example, a `java.lang.String` class can be mapped to either an `xsd:string` or `soapenc:string`. The **Java2WSDL** command chooses one of these mappings, but the WSDL file must be edited if a different mapping is desired.
- There are multiple ways to generate WSDL constructs. For example, the part element in the `wsdl:message` can be generated with a `type` or `element` attribute. The **Java2WSDL** command makes an informed choice based on the settings of the -style and -use options.
- The WSDL file describes the instance data elements sent in the SOAP message. If you want to modify the names or format used in the message, the WSDL file must be edited. For example, write a bean property value as an attribute instead of an element.
- The WSDL file requires editing if header or attachment support is desired.
- The WSDL file requires editing if a multipart WSDL, one using `wsdl:import`, is desired.

For simple services, the generated WSDL file is sufficient. For complicated services, the generated WSDL file is a good starting point.

General issues
- **Package to namespace mapping**

The JAX-RPC specification does not specify the default mapping of Java package names to XML namespaces. The JAX-RPC specification does specify that each Java package must map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm is provided that constructs the namespace by reversing the names of the Java package and adding an `http://` prefix. For example, a package named, `com.ibm.webservice`, is mapped to the namespace `http://webservice.ibm.com`.

The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of the **WSDL2Java** and **Java2WSDL** commands.

- **Identifier mapping**

  Java identifiers are mapped directly to WSDL file and XML identifiers.

  Java bean property names are mapped to the WSDL file and XML identifiers. For example, a Java bean, with getInfo and setInfo methods, maps to an XML construct with the name, `info`.

  The Service Endpoint Interface method parameter names, if available, are mapped directly to the XML identifiers. See the WSDL2Java command -implClass option for more details.
- **WSDL construction summary**

  The following table summarizes the mapping from a Java construct to the related WSDL and XML construct.

Java construct	WSDL and XML construct
Service Endpoint Interface	`wsdl:portType`
Method	`wsdl:operation`
Parameters	`wsdl:input, wsdl:message, wsdl:part (1)`
Return	`wsdl:output, wsdl:message, wsdl:part (1)`
Throws	`wsdl:fault, wsdl:message, wsdl:part (1)`
Primitive types	`xsd` and `soapenc` simple types
Java beans	`xsd:complexType`
Java bean properties	Nested `xsd:elements` of `xsd:complexType`
Arrays	JAX-RPC defined array `xsd:complexType`
User defined exceptions	`xsd:complexType`

Note: The generated WSDL file is affected by the -style and -use options. A `wsdl:binding` that conforms to the generated `wsdl:portType` is generated. The style and use constructs of the `wsdl:binding` are determined from the -style and -use options. A `wsdl:service` containing a port that references the generated `wsdl:binding` is generated. The names and values of the `wsdl:service` are controlled by the **Java2WSDL** command options.

- **Style** and **use**

  Use the -style and -use options to generate different kinds of WSDL files. The four supported combinations are:
  - -style RPC -use ENCODED
  - -style DOCUMENT -use LITERAL
  - -style RPC -use LITERAL
  - -style DOCUMENT -use LITERAL -wrapped false

  The -use LITERAL option affects the generated WSDL file in the following ways:
  - The `soap:body` elements in the `wsdl:binding` are specified as `use="literal"`.

- The soap:fault elements in the wsdl:binding are specified as use="literal".
- The soap encoded types are not used.
- The soap encoded array style is not used. The maxOccurs attribute is used to indicate arrays.

The -use ENCODED option affects the generated WSDL file in the following ways:
- The soap:body elements in the wsdl:binding are specified as use="encoded" and the encodingStyle is set.
- The soap:fault elements in the wsdl:binding are specified as use="encoded" and the encodingStyle is set.

The -style RPC option affects the generated WSDL file in the following ways:
- The wsdl:part elements use the type attribute to reference XML types.
- The wsdl:binding is specified as style="rpc".

The -style DOCUMENT -wrapped false option affects the generated WSDL file in the following ways:
- The wsdl:part elements use the type attribute to reference simple types. The element attribute is used to reference the root xsd:elements for everything that is not a simple type.
- The wsdl:binding is specified as style="document".

The -style DOCUMENT -wrapped true option generates a WSDL file that conforms to the .NET WSDL file format:
- A request xsd:element is generated for each method in the Service Endpoint Interface as follows:
  - The name of the xsd:element is the same as the name of the wsdl:operation.
  - The xsd:element contains an xsd:sequence that contains xsd:elements defining each parameter.
  - The request wsdl:message references the wrapper xsd:element using a single part:
    - The name of the part is parameters.
    - The element attribute is used to reference the wrapper xsd:element.
- A response xsd:element is generated for each method in the Service Endpoint Interface as follows:
  - The name of the xsd:element is the same as the name of the wsdl:operation appended with Response
  - The xsd:element contains an xsd:sequence that contains xsd:elements defining the return value.
  - The request wsdl:message references this wrapper xsd:element using a single part.
    - The element attribute is used to reference the wrapper xsd:element.
- The wsdl:binding is specified as style="document".

Mapping of standard XML types from Java types

Some Java types map directly to standard XML types. These types do not require additional XML definitions in the wsdl:types section.

- **JAX-RPC Java primitive type mapping**

  The following table describes the mapping from the Java primitive and standard types to XML standard types. For more information see the JAX-RPC specification.

Java type	XML type

int	xsd:int
long	xsd:long
short	xsd:short
float	xsd:float
double	xsd:double
boolean	xsd:boolean
byte	xsd:byte
byte[]	xsd:base64Binary **Note:** The default mapping for byte[] is xsd:base64Binary. The data in byte[] is passed over the wire as a text string encoded in the base64 format. An alternative format is xsd:hexBinary. To use the xsd:hexBinary format: • Edit the WSDL file and change xsd:base64Binary to xsd:hexBinary.
java.lang.String	xsd:string
java.math.BigInteger	xsd:integer
java.math.BigDecimal	xsd:decimal
java.util.Calendar	xsd:dateTime
java.util.Date **Note:** This mapping is not covered by the JAX-RPC.	xsd:date
java.lang.Boolean	xsd:boolean xsi:nillable=true
java.lang.Float	xsd:float xsi:nillable=true
java.lang.Double	xsd:double xsi:nillable=true
java.lang.Long	xsd:long xsi:nillable=true
java.lang.Integer	xsd:int xsi:nillable=true
java.lang.Short	xsd:short xsi:nillable=true
java.lang.Byte	xsd:byte xsi:nillable=true

Note: The java.lang wrapper classes in the last seven lines of the table map to the same XML construct as the corresponding Java primitive type. In addition, the xsi:nillable attribute is generated to indicate that such elements can be null.

- **Additional Java class mappings**

  In addition to the standard JAX-RPC mapping, the following classes are mapped directly to XML types:

Java type	XML type
java.util.Map **Note:** Any classes that implement java.util.Map are also mapped to apache:Map.	apache:Map
java.util.Collection **Note:** Each Java array, except byte[], and every class that implements java.util.Collection is mapped to a JAX-RPC defined soapenc:Array type.	soapenc:Array
org.w3c.dom.Element	apache:Element
java.util.Vector	apache:Vector

`java.awt.Image` **Note:** Used for attachment support.	`apache:Image`
`javax.mail.internet.MimeMultiPart` **Note:** Used for attachment support.	`apache:Multipart`
`javax.xml.transform.Source` **Note:** Used for attachment support.	`apache:Source`
`javax.activation.DataHandler` **Note:** Used for attachment support.	`apache:DataHandler`

Generation of wsdl:types

Java types that cannot be mapped directly to standard XML types are generated in the `wsdl:types` section.

- **Java arrays**

  Java arrays for the -use ENCODED option, with the exception of `byte[]`, are generated using the following format. See the JAX-RPC specification for more details. Alternative mappings can be found in Table 18.1 of the JAX-RPC specification.

  Java:

  ```
  Item[]
  ```

  Mapped to:

  ```
  <xsd:complexType name="ArrayOfItem">

   <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
     <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="ns:Item" />
    </xsd:restriction>
    </xsd:complexContent
  </xsd:complexType>
  ```

- **JAX-RPC value type and bean mapping**

  A Java class that matches the Java value type or bean pattern is mapped to an `xsd:complexType`. In order for a Java class to be mapped to XML, follow these conditions:
  - The class must have a public default constructor.
  - The class must not implement, directly or indirectly, `java.rmi.Remote`.
  - Public, non-static, non-final, non-transient fields are mapped. The class can contain other fields and methods, but they are not mapped.
  - If the class follows the Java bean pattern and has public getter and setter methods, the property is mapped.

  Additional mapping rules can be found in the JAX-RPC specification. The following example indicates the mapping for sample base and derived Java classes:

  Java:

  ```
  public abstract class Base {
       public Base() {}
       public int a;                        // mapped
       private int b;                       // mapped via setter/getter
       private int c;                       // not mapped
       private int[] d;                    // mapped via indexed setter/getter
  ```

```
   public int getB() { return b;}        // map property b
   public void setB(int b) {this.b = b;}

   public int[] getD() { return d;}      // map indexed property d
   public void setD(int[] d) {this.d = d;}
   public int getD(int index) { return d[index];}
   public void setB(int index, int value) {this.d[index] = value;}

   public void someMethod() {...}        // not mapped
}

public class Derived extends Base {
   public int x;                         // mapped
   private int y;                        // not mapped
}
```

Mapped to:

```
<xsd:complexType name="Base" abstract="true">
 <xsd:sequence>
  <xsd:element name="a" type="xsd:int" />
  <xsd:element name="b" type="xsd:int" />
 <xsd:element name="d" minOccurs="0" maxOccurs="unbounded" type="xsd:int"/>
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
 <xsd:complexContent>
  <xsd:extension base="ns:Base">
   <xsd:sequence>
    <xsd:element name="x" type="xsd:int" />
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Inheritance and abstract classes

The example contains two optional JAX-RPC features that are supported by WebSphere Application Server:

1. An abstract class is mapped to an `xsd:complexType` with `abstract="true"`.
2. An indexed bean property (see the methods for d in Base) are mapped to a nested element specified with `maxOccurs="unbounded"`. This format is similar to an XML array, but the SOAP message is different. An XML array defines an element for the array and nested elements for each item in the array. An element defined with `maxOccurs` indicates a series of items without the surrounding array wrapper element. Both formats are popular.

- **JAX-RPC enumeration class mapping**

  Section 4.2.4 of the JAX-RPC specification defines the mapping from an XML enumeration to a Java class. Though not specifically required by the JAX-RPC, the **Java2WSDL** command performs the reverse mapping. If you have a class that has the same format as a JAX-RPC enumeration class, it is mapped to an XML enumeration.
- **Holder classes**

  The JAX-RPC specification defines Holder classes in section 4.3.5. A Holder class is used to support `in` and `out` parameter passing. Every Holder class implements the`javax.xml.rpc.holders.Holder` interface. The **Java2WSDL** command maps Holder classes to the same XML type as the held type. In addition, references to Holder classes affect the generation of `wsdl:messages`.
- **Exception classes**

If a class extends the exception, `java.lang.Exception`, it is mapped to an `xsd:complexType` similar to the Java bean mapping. The getter methods of the exception are mapped as nested `xsd:elements` of the `xsd:complexType`. See section 5.5.5 of the JAX-RPC specification for more details.

Note: You need to generate implementation specific exception classes by invoking the **WSDL2Java**command on the resulting WSDL file.

- **Unsupported classes**

  If a class cannot be mapped to an XML type, the **Java2WSDL** command issues a message and an `xsd:anyType` reference is generated in the WSDL file. In these situations, modify the Web service implementation to use the JAX-RPC compliant classes.
- **Generation of root elements**

  If the **Java2WSDL** command generates an `xsd:complexType` or `xsd:simpleType` for a parameter reference, a corresponding `xsd:element` is also generated. The `xsd:element` has the same name as the `xsd:complexType/xsd:simpleType` and uses the `type` attribute to reference the `xsd:complexType/xsd:simpleType`. The `wsdl:message` part can use the `element` attribute or the `type` attribute to reference the `xsd:element` or type. This choice is determined by the -style and -use options.

Generation from the interface or implementation class

The class passed to the **Java2WSDL** command represents the interface of the `wsdl:service`. The `wsdl:portType` and `wsdl:message` elements generate from this interface or implementation class.

- **Generation of the wsdl:portType**

  The name of the `wsdl:portType` is the name of the class unless overridden by the -portTypeName option.
- **Generation of wsdl:operation**

  A `wsdl:operation` generates for each public method in the interface that throws the exception, `java.rmi.RemoteException`.
  - The name of the `wsdl:operation` is the name of the method.
  - The `wsdl:operation` has a `parameterOrder` attribute, which defines the order of the parameters in the signature. Specifically, the `parameterOrder` lists the order of the parts of the request or response `wsdl:messages`.
  - The `wsdl:operation` has a nested `wsdl:input` element that references the request `wsdl:message` using the `message` attribute.
  - The `wsdl:operation` has a nested `wsdl:output` element that references the response `wsdl:message` using the `message` attribute.
  - The `wsdl:operation` has a nested `wsdl:fault` element that references the default `wsdl:message` using the `message` attribute.

  See sections 5.5.4 and 5.5.5 of the JAX-RPC specification for more information.
- **Generation of wsdl:message**

  Generating the `wsdl:message` is directly related to the -style and -use options. The following is the default mapping (-style RPC -use ENCODED):
  - A `wsdl:message` is created to represent the request.
    - A `wsdl:part` representing each parameter is added to the `wsdl:message`.
      - The `wsdl:part` has the same name as the parameter.
      - The `wsdl:part` uses the `type` attribute to locate the XML type of the parameter.
  - A `wsdl:message` is created to represent the response:
    - A `wsdl:part` representing the method return is created.
      - The `wsdl:part` has the same name as the method with Return appended.

Note:
The name of the part is not specified by the JAX-RPC and is typically not checked by SOAP engines.
- The wsdl:part has the same name as the parameter.
- The wsdl:part uses the type attribute to locate the XML type of the parameter.
- A wsdl:part is created for each parameter that is a Holder.
- The wsdl:part has the same name as the parameter.
- A wsdl:message is created to represent the fault if the operation has a wsdl:fault.
- A wsdl:part representing the fault is created.
- The wsdl:part has the same name as the exception.
- The wsdl:part uses the type attribute to locate the complexType representing the exception.

The same mapping is used as described if you use the -style RPC and -use LITERAL options. It is also valid to use the wsdl:part element attribute instead of the type attribute to reference the XML schema. If you use the -style DOCUMENT -wrapped false and -use LITERAL options, the same mapping is used as described except the wsdl:part element attribute is used to reference the XML schema. If the XML schema is a primitive type, like xsd:string, the type attribute is used to reference the XML type. The -style DOCUMENT, -wrapped true and -use LITERAL options result in completely different mappings for the request and response messages. For example:

– A request xsd:element is generated for each method in the Service Endpoint Interface.
  - The name of the xsd:element is the same as the name of the wsdl:operation.
  - The xsd:element contains an xsd:sequence that contains xsd:elements defining each parameter.
  - The request wsdl:message references the wrapper xsd:element using a single part.
    - The name of the part is parameters.
    - The element attribute is used to reference the wrapper xsd:element.
– A response xsd:element is generated for each method in the Service Endpoint Interface.
  - The name of thexsd:element is the same as the name of the wsdl:operation appended with Response.
  - The xsd:element contains an xsd:sequence that contains xsd:elements defining the return value.
  - The request wsdl:message references this wrapper xsd:element using a single part.
    - The element attribute is used to reference the wrapper xsd:element.

- **Generation of wsdl:binding**

  Generate a wsdl:binding with a name defined by the **Java2WSDL -bindingName** command.
  – The wsdlsoap:binding style attribute is set to rpc if you use the -style RPC option; otherwise it is set to document.
  – A wsdl:operation generates for each wsdl:operation defined in the wsdl:portType.
  – Each wsdl:operation has corresponding wsdl:input, wsdl:outputand wsdl:fault elements.
  – The wsdl:input, wsdl:output and wsdl:fault elements each contain a wsdlsoap:body element.

- The wsdlsoap:body use attribute is set to literal or encoded according to the **-use** argument. Set the encodingStyle attribute to http://schemas.xmlsoap.org/soap/encoding/ when **use** is encoded.
- **Generation of the wsdl:service**

  Generate a wsdl:service with a name defined by the **Java2WSDL -serviceElement** command. For example:
  - The wsdl:service contains a port with a name defined by the **Java2WSL -servicePortName** command.
  - The port references the generated wsdl:binding with the binding attribute.
  - The port contains a wsdlsoap:address element with a
  - The location attribute is set to the value of the **Java2WSDL -location** command.

WSDL-to-Java mapping

The **WSDL2Java** command tool uses the following rules to generate Java classes when developing your Web services client and server. In addition, implementation specific Java classes are generated that assist in the serialization and deserialization, and invocation of the Web service.

General issues
- **Mapping of namespace to package**

  The JAX-RPC does not specify the mapping of XML namespaces to Java package names. The JAX-RPC does specify that each Java package map to a single XML namespace. Likewise, each XML namespace must map to a single Java package. A default mapping algorithm omits any protocol from the XML namespace and reverses the names. For example, an XML namespace http://websphere.ibm.com becomes a Java package with the name com.ibm.websphere.

  The default mapping of XML namespace to Java package disregards the context-root. If two namespaces are the same up until the first slash, they map to the same Java package. For example, the XML namespaces http://websphere.ibm.com/foo and http://websphere.ibm.com/bar map to the Java package com.ibm.websphere. The default mapping between XML namespaces and Java package names can be overridden using the -NStoPkg and -PkgtoNS options of **WSDL2Java** and **Java2WSDL** commands.
- **Identifier mapping**

  XML names are much richer than Java identifiers. They can include characters that are not permitted in Java identifiers. See section 20 of the JAX-RPC specification for the rules to map an XML name to a Java identifier.

  The mapping rules attempt to follow accepted Java coding conventions. Class names always begin with an uppercase letter. Method names begin with a lowercase letter. The **WSDL2Java**command generates metadata in the _Helper class so that the values are serialized or deserialized using the XML names specified in the WSDL file.
- **Java construction summary**

WSDL and XML	Java
xsd:complexType (struct) **Note:** The xsd:complexType can also represent a Java exception if referenced by a wsdl:message for a wsdl:fault .	Java Bean Class **Note:** The classes, _Helper, _Ser, and _Deser, generate for each Java bean class. These implementation classes aid serialization and deserialization.
nested xsd:element/xsd:attribute	Java bean property
xsd:complexType (array)	Java array

xsd:simpleType (enumeration)	JAX-RPC enumeration class
xsd:complexType (wrapper)The method parameter signature typically is determined by the wsdl:message. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper xsd:element	Service Endpoint Interface method parameter signature **Note:** If a parameter is out or inout, a Holder class generates.
--	--
wsdl:messageThe method parameter signature typically is determined by the wsdl:message. However, if the WSDL file is a .NET wrapped style, the method parameter signature is determined by the wrapper xsd:element	Service Endpoint Interface method signature **Note:** If a parameter is out or inout, a Holder class generates.
wsdl:portType	Service Endpoint Interface
wsdl:operation	Service Endpoint Interface method
wsdl:binding	Stub **Note:** The Stub and ServiceLocator classes are implementation specific.
wsdl:service	Service Interface and ServiceLocator **Note:** The Stub and ServiceLocator classes are implementation specific.
wsdl:port	Port accessor method in Service Interface

Mapping standard XML types

- **JAX-RPC simple XML types mapping**

  The following mappings are XML types to Java types. For more information about these mappings, see section 4.2.1 of the JAX-RPC specification.

XML type	Java type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	int
xsd:long **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	long
xsd:short **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	short
xsd:decimal	java.math.BigDecimal
xsd:float **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	float
xsd:double **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	double

xsd:boolean **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	boolean
xsd:byte **Note:** If an element with this type has the xsi:nillable attribute set to true, it is mapped to the Java wrapper class of the primitive type.	byte
xsd:dateTime	java.util.Calendar
xsd:date **Note:** This mapping is not supported by the JAX-RPC.	java.util.Date
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]
--	--
soapenc:base64	byte[]
soapenc:base64Binary	byte[]
soapenc:string	java.lang.String
soapenc:boolean	java.lang.Boolean
soapenc:float	java.lang.Float
soapenc:double	java.lang.Double
soapenc:decimal	java.math.BigDecimal
soapenc:int	java.lang.Integer
soapenc:integer **Note:** This mapping is not supported by the JAX-RPC.	java.math.BigInteger
soapenc:short	java.lang.Short
soapenc:long **Note:** This mapping is not supported by the JAX-RPC.	java.lang.Long
soapenc:byte	java.lang.Byte

- **JAX-RPC optional simple XML type mapping**

  The **WSDL2Java** command supports the following JAX-RPC optional simple XML types.

XML type	Java type
xsd:qname	javax.xml.namespace.QName

- **JAX-RPC xsd:anyType mapping**

  The **WSDL2Java** command maps an `xsd:anyType` to a `java.lang.Object`. This is an optional feature of the JAX-RPC specification. The `xsd:anyType` can be used to store any XML type other than the XML primitive type. An `xsd:anyType` is always serialized along with an `xsi:type` that specifies the actual type.
- **Additional supported mappings**

  The following mappings are also supported by the **WSDL2Java** command. These mappings are not defined by the JAX-RPC specification.

XML type	Java type
apache:PlainText **Note:** For MIME attachments.	java.lang.String
apache:Map	java.util.Map

`apache:Element`	`org.w3c.dom.Element`
`apache:Vector`	`java.util.Vector`
`apache:Image` **Note:** For MIME attachments.	`java.awt.Image`
`apache:Multipart` **Note:** For MIME attachments.	`javax.mail.internet.MimeMultipart`
`apache:Source` **Note:** For MIME attachments.	`javax.xml.transform.Source`
`apache:octetStream` **Note:** For MIME attachments.	`javax.activation.DataHandler`
`apache:DataHandler` **Note:** For MIME attachments.	`javax.activation.DataHandler`

Mapping XML defined in the `wsdl:types` section

The **WSDL2Java** command generates Java types for the XML schema constructs defined in the `wsdl:types` section. The XML schema language is broader than the required or optional subset defined by the JAX-RPC specification. The **WSDL2Java** command supports all required mappings and most optional mappings. In addition, the command supports some XML schema mappings that are outside the JAX-RPC specification. In general, the **WSDL2Java** command ignores constructs that it does not support. For example, the **WSDL2Java** command does not support the `default` attribute. If an `xsd:element` is defined with the `default` attribute, the `default` attribute is ignored. In some cases it maps unsupported constructs to `wasws:SOAPElement`.

- **Mapping of xsd:complexType to Java bean**

  The most common mapping is from an `xsd:complexType` to a Java bean class.
  - *Standard Java bean mapping*

    The standard Java bean mapping is defined in section 4.2.3 of the JAX-RPC specification The `xsd:complexType` defines the type. The nested `xsd:elements` within the `xsd:sequence` or `xsd:all` groups are mapped to Java bean properties. For example:

    XML:

```
<xsd:complexType name="Sample">
  <xsd:sequence>
  <xsd:element name="a" type="xsd:string"/>
  <xsd:element name="b" maxOccurs="unbounded" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
```

Java:

```
public class Sample {
     // ..
     public Sample() {}

     // Bean Property a
     public String getA()             {...}
     public void   setA(String value) {...}

     // Indexed Bean Property b
     public String[] getB()           {...}
     public String   getB(int index)  {...}
```

```
    public void     setB(String[] values) {...}
    public void     setB(int index, String value) {...}

}
```

- ***Methods equals() and hashCode()***

  The generated Java bean classes contain an implementation of the equals() method. The generation of this method is outside the JAX-RPC specification. The equals() method returns `true` if equals() is `true` for each contained bean property. The implementation accounts for self-referencing loops. This version of the equals() method is typically more useful than the "identity" equals provided by `java.lang.Object`.

  A corresponding hashCode() method is also generated in the Java bean class.
- ***Properties and indexed properties***

  In the standard Java bean mapping example, the nested `xsd:element` for property `a` is mapped to a Java bean property. In addition, the **WSDL2Java** command maps a nested `xsd:element` with `maxOccurs` > 1 to a Java bean indexed property.
- ***Attributes***

  The **WSDL2Java** command also supports the `xsd:attribute` element, as shown in the following example.

  Attribute `a` is mapped as a Java bean property, which is exactly the same mapping as a nested `xsd:element`. Implementation specific metadata is generated in the `Sample2_Helper` class to ensure that property `a` is serialized and deserialized as an attribute, and not as a nested element. For example:

  XML:

```
<xsd:complexType name="Sample2">
  <xsd:sequence>
   <xsd:attribute name="a" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

  Java:

```
public class Sample2 {
     // ..
     public Sample2() {}

     // Bean Property a
     public String getA()            {...}
     public void   setA(String value) {...}
  }
```

- ***Qualified versus unqualified names***

  The **WSDL2Java** command supports the `elementForm` and `attributeForm` schema attributes.

  This support is not specified in the JAX-RPC specification. These attributes are used to indicate whether an element or attribute is serialized and deserialized with a qualified or unqualified name. The default setting for `elementForm` is `qualified` and the default setting for `attributeForm` is `unqualified`. These settings do not affect the Java bean class that is generated, but the information is captured in the `_Helper` class metadata.
- ***Extension and the abstract attribute***

  The **WSDL2Java** command supports extension of an `xsd:complexType` through the `xsd:extension` element. This support is required by the JAX-RPC specification.

The **WSDL2Java** command supports the `abstract` attribute. This feature is listed as optional by the JAX-RPC specification.

The following example shows the accepted use of the extension and abstract constructs. WebSphere Application Server uses the extension and abstract constructs to support polymorphism.

XML:

```
<xsd:complexType name="Base" abstract="true">
 <xsd:sequence>
  <xsd:element name="a" type="xsd:int" />
 </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Derived">
 <xsd:complexContent>
  <xsd:extension base="ns:Base">
   <xsd:sequence>
    <xsd:element name="b" type="xsd:int" />
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Java:

```
   public abstract class Base {
      // ...
      public Base() {}

      public int getA() {...}
      public void setA(int a) {...}
   }

   public class Derived extends Base {
      // ...
      public Derived() {}

      public int getB() {...}
      public void setB(int b) {...}
   }
```

– *Support for* ***xsd:any***

The **WSDL2Java** command supports `xsd:anyelement`, which is different than `xsd:anyType`. This feature is not defined within the JAX-RPC specification and is subject to change.

If an `<xsd:any/>` element is defined within `xsd:sequence` or `xsd:all group`, SOAP values that do match one of the `xsd:elements` are stored in the Java bean. Values can be accessed from the Java bean using the get_any() and set_any() methods.

- **Mapping of xsd:element**

  An `xsd:element` is a construct that has a name or `name` attribute, and a type defined by a `complexType` or `primitive type`. There are two different kinds of `xsd:elements`:

  – **Root:** Defined directly underneath the schema elements and referenced by other constructs.

  – **Nested:** Nested underneath group elements and are not referenced by other constructs.

Root elements are referenced by the WSDL file constructs, especially if the WSDL file is used to describe a literal service. Typically, root elements and types have the same names, which is allowed in the schema language. Under most circumstances the **WSDL2Java** command can produce Java artifacts without name collisions.

– ***Four ways to reference a type***

  There are four ways that a nested or root `xsd:element` can reference a type:

  - **Use the `type` attribute:**

    This is the most common way to reference a type, for example:

    ```
    <xsd:element name="one" type="ns:myType" />
    ```

    The **WSDL2Java** command recognizes the `type` attribute as a reference to a `complexType` or `simpleType`named, `myType`. The **WSDL2Java** command generates a Java type based on the characteristics of `myType`. Support for the `type` attribute is required by the JAX-RPC specification.

  - **Use the `ref` attribute:** For example:

    ```
    <xsd:element ref="ns:myElement" />
    ```

    The **WSDL2Java** command recognizes the `ref` attribute as a reference to another root element named `myElement`. The name of the element is obtained from the referenced element, such as `myElement`. The type of the element is the type of the referenced element. The **WSDL2Java** command generates a Java type based on the characteristics of the referenced type. The `ref` attribute is an optional feature of the JAX-RPC specification.

  - **Use no attribute:**

    For example:

    ```
    <xsd:element name="three" />
    ```

    When you do not use an attribute, the **WSDL2Java** command recognizes a reference to the `xsd:anyType` as defined by the XML schema specification. The `xsd:anyType` is an optional type of the JAX-RPC specification.

  - **Use an anonymous type:**

    For example:

    ```
    <xsd:element name="four">
      <xsd:complexType>
       <xsd:sequence>
        <xsd:element name="foo" type="xsd:string" />
       </xsd:sequence>
      </xsd:complexType>
     </ xsd:element>
    ```

    When you use an anonymous type, the **WSDL2Java** command recognizes a reference to the type defined within the element.

    Note: The `complexType` does not have a name.

    The **WSDL2Java** command generates a Java type based on the characteristics of this type. Since the anonymous type does not have a name, the **WSDL2Java** command uses the name of the `container` element, which can result in collisions with defined types and other anonymous types. The **WSDL2Java** command automatically detects and renames classes to avoid collisions. Support for anonymous types is not defined by the JAX-RPC specification, however using anonymous types is common.

Note: An `xsd:attribute` is like an `xsd:element`; it contains a name and refers to a type. An `xsd:attribute` can refer to its type with the `type`attribute or using an anonymous type.

– ***Element specific attributes***

Some attributes can be applied to `xsd:elements` and not to XML types.

The `maxOccurs` attribute indicates the maximum number of occurrences of the element in the SOAP message. The default value is 1. If the value is greater than 1, or unbounded, the **WSDL2Java** command maps the construct to a Java array or bean indexed property. Metadata is also generated to properly serialize and deserialize a series of elements versus a normal XML array. The `maxOccurs` attribute is an optional feature of the JAX-RPC specification.

The `minOccurs` attribute indicates the minimum number of occurrences of the element in the SOAP message. The default value is 1. The `xsi:nillable` attribute indicates whether the element can have a `nil` value. The `minOccurs` and `xsi:nillable` settings affect how a `null` value is serialized in a SOAP message. If `minOccurs`=0, the `null` value is not serialized. If `xsi:nillable=true`, the value is serialized with the `xsi:nil=true` attribute.

- **Mapping of xsd:complexType to Java array**

  The **WSDL2Java** command maps the following three kinds of XML formats to Java arrays:

  XML:

```
<xsd:element name="array1" type="soapenc:Array" />
```

  Java:

```
Object[] array1;
```

  XML:

```
<xsd:complexType name="arrayOfInt">
 <xsd:complexContent>
  <xsd:restriction base:"soapenc:Array">
   <xsd:attribute ref:"soapenc:arrayType" wsdl:arrayType="xsd:int[]" />
  </xsd:restriction>
  </xsd:complexContext>
 </xsd:complexType>

 <xsd:element name="array2" type="ns:arrayOfInt" />
```

  Java:

```
  int[] array2;
```

  XML:

```
<xsd:complexType name="arrayOfInt">
 <xsd:complexContent>
  <xsd:restriction base:"soapenc:Array">
   <xsd:sequence>
    <xsd:element name="item" type="xsd:int" maxOccurs="unbounded" />
   </xsd:sequence>
  </xsd:restriction>
 </xsd:complexContent>
</xsd:complexType>

 <xsd:element name="array3" type="ns:arrayOfInt" />
```

  Java:

```
  int[] array3;
```

- **Mapping of xsd:simpleType enumeration**

The **WSDL2Java** command maps the following XML enumeration to a JAX-RPC specified enumeration class. See section 4.2.4 of the JAX-RPC specification for more details.

```
<xsd:simpleType name="EyeColorType" >

 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="brown"/>
  <xsd:enumeration value="green"/>
  <xsd:enumeration value="blue"/>
 </xsd:restriction>
</xsd:simpleType>
```

- **Mapping of xsd:complexType to exception class**

  If a `complexType` is referenced in a `wsdl:message` for a `wsdl:fault`, the `complexType` is mapped to a class that extends the exception, `java.lang.Exception`. This mapping is similar to the mapping of a `complexType` to a Java bean class, except a full constructor is generated, and only getter methods are generated. See section 4.3.6 of the JAX-RPC specification for more details.
- **Other mappings**

  The **WSDL2Java**command supports the mapping of `xsd:simpleType` and `xsd:complexTypes` that extend `xsd:simpleTypes`. These constructs are mapped to Java bean classes. The simple value is mapped to a Java bean property named, `value`. This is an optional feature of the JAX-RPC specification.

Mapping of `wsdl:portType`

The `wsdl:portType` construct is mapped to the Service Endpoint Interface. The name of the `wsdl:portType` is mapped to the name of the class of the Service Endpoint Interface.

Mapping of `wsdl:operation`

A `wsdl:operation` within a `wsdl:portType` is mapped to a method of the Service Endpoint Interface. The name of the `wsdl:operation` is mapped to the name of the method. The `wsdl:operation` contains `wsdl:input` and `wsdl:output` elements that reference the request and response `wsdl:message` constructs using the `message` attribute. The `wsdl:operation` can contain a `wsdl:fault` element that references a `wsdl:message` describing the fault. These faults are mapped to Java classes that extend the exception, `java.lang.Exception` as discussed in section 4.3.6 of the JAX-RPC specification.

- **Effect of document literal wrapped format**

  If the WSDL file uses the .NET document and literal wrapped format, the method parameters are mapped from the wrapper `xsd:element`. The .NET document and literal format is automatically detected by the **WSDL2Java** command. The following criteria must be met:
  - The WSDL file must have `style="document"` in its `wsdl:binding` constructs.
  - The WSDL file must have `use="literal"` in its `wsdl:binding` constructs.
  - The `wsdl:message` referenced by the `wsdl:operation` input construct must have a single part.
  - The part must use the `element` attribute to reference an `xsd:element`.
  - The referenced `xsd:element`, or wrapper element, must have the same name as the `wsdl:operation`.
  - The wrapper element must not contain any `xsd:attributes`.

In such cases, each parameter name is mapped from a nested `xsd:element` contained within wrapper element. The type of the parameter is mapped from the type of the nested `xsd:element`. For example:

XML:

```
<xsd:element name="myMethod" >
 <xsd:complexType>
  <xsd:sequence>
   <xsd:element name="param1" type="xsd:string" />
   <xsd:element name="param2" type="xsd:int" />
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>
...
<wsdl:message name="response" />
 <part name="parameters" element="ns:myMethod" />
</wsdl:message name="response" />

<wsdl:message name="response" />
...
<wsdl:operation name="myMethod">
 <input name="input" message="request" />
 <output name="output" message="response" />
</wsdl:operation>
```

Java:

```
void myMethod(String param1, int param2) ...
```

- **Parameter mapping**

  If the document and literal wrapped format is not detected, the parameter mapping follows the normal JAX-RPC mapping rules set in section 4.3.4 of the JAX-RPC specification.

  Each parameter is defined by a `wsdl:message` part referenced from the `input` and `output` elements.
  - A `wsdl:part` in the request `wsdl:message` is mapped to an `input` parameter.
  - A `wsdl:part` in the response `wsdl:message` is mapped to the `return` value. If there are multiple `wsdl:parts` in the response message, they are mapped to `output` parameters.
    - A `Holder` class is generated for each `output` parameter as discussed in section 4.3.5 of the JAX-RPC specification.
  - A `wsdl:part` that is both the request and response `wsdl:message` is mapped to an `inout` parameter.
    - A `Holder` class is generated for each `inout` parameter as discussed in section 4.3.5 of the JAX-RPC specification.
    - The `wsdl:operation` `parameterOrder` attribute defines the order of the parameters.

The **WSDL2Java** command supports overloaded methods, but confirm that the part names of the overloaded methods are unique. For example:

XML:

```
<wsdl:message name="request" >
 <part name="param1" type="xsd:string" />
 <part name="param2" type="xsd:int" />
</wsdl:message name="response" />

<wsdl:message name="response" />
...
<wsdl:operation name="myMethod" parameterOrder="param1, param2">
 <input name="input" message="request" />
 <output name="output" message="response" />
```

```
</wsdl:operation>


Java:

void myMethod(String param1, int param2) ...
```

Mapping of wsdl:binding

The **WSDL2Java** command uses the wsdl:binding information to generate an implementation specific client side stub. WebSphere Application Server uses the wsdl:binding information on the server side to properly deserialize the request, invoke the Web service, and serialize the response. The information in the wsdl:binding should not affect the generation of the Service Endpoint Interface, but it can when the document and literal wrapped format is used or when there are MIME attachments.

- **MIME attachments**

  For a WSDL 1.1 compliant WSDL file, a part of an operation message, which is defined in the binding to be a MIME attachment, becomes a parameter of the type of the attachment regardless of the part declared. For example:

```
XML:
<wsdl:types>
 <schema ...>
  <complexType name="ArrayOfBinary">
   <restriction base="soapenc:Array">
     <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:binary[]" />
   </restriction>
  </complexType>
 </schema>
</wsdl:types>

<wsdl:message name="request">
 <part name="param1" type="ns:ArrayOfBinary" />
<wsdl:message name="response" />

<wsdl:message name="response" />
 ...

 <wsdl:operation name="myMethod">
  <input name="input" message="request" />
  <output name="output" message="response" />
 </wsdl:operation>
  ...

<binding ...
 <wsdl:operation name="myMethod">
  <input>
   <mime:multipartRelated>
    <mime:part>
     <mime:content part="param1" type="image/jpeg"/>
    </mime:part>
   </mime:multipartRelated>
  </input>
   ...
 </wsdl:operation>

Java:

void myMethod(java.awt.Image param1) ...
```

The JAX-RPC requires support for the following MIME types:

MIME type	Java type

image/gif	`java.awt.Image`
image/jpeg	`java.awt.Image`
text/plain	`java.lang.String`
multipart/*	`javax.mail.internet.MimeMultipart`
text/xml	`javax.xml.transform.Source`
application/xml	`javax.xml.transform.Source`

There are a number of problems with MIME attachments as they are defined in WSDL 1.1, including:

– The semantics of the `mime:multipartRelated` clause are not fully defined
– The semantics do not allow for arrays of MIME attachments

Because of these problems, several types are not specified by the JAX-RPC for MIME attachments. These types are defined in the supported mappings previously discussed.

- **Headers**

  A `wsdl:binding` can also define SOAP headers, for example:

  XML:

```
<wsdl:message name="request">
  <part name="param1" type="xsd:string" />
 </wsdl:message/>

 <wsdl:message name="response" />
...

 <wsdl:operation name="myMethod">
  <input name="input" message="request" />
  <output name="output" message="response" />
 </wsdl:operation>
 ...

<binding ...
 <wsdl:operation name="myMethod">
  <input>
   <soap:header message="request" part="param1" use="literal" />
  </input>
  ...
 </wsdl:operation>
```

  Java:

```
void myMethod(String param1) ...
```

  This is an example of an explicit header or a header with a value determined from a `method` parameter. Instead of appearing in the `soap:body` SOAP message, the value of `param1` now appears in the `soap:header` SOAP message. The **WSDL2Java** command supports explicit headers and does not support implicit headers. Implicit headers have a value not determined by a parameter. For example, you could replace the `soap:header` clause in the example with:

```
<soap:header message="someOtherMsgNotAppearingInthePortType"
  part="someOtherPart" use="literal"/>
```

  Note: The **WSDL2Java** command supports explicit headers, but it is not considered good programming practice to use them. Headers are typically used for middleware logic, not business logic. Explicit headers place parameters used in business logic into the header.

Mapping of wsdl:service

The wsdl:service element is mapped to a Generated Service interface. The Generated Service interface contains methods to access each of the ports in the wsdl:service. The Generated Service interface is discussed in sections 4.3.9, 4.3.10, and 4.3.11 of the JAX-RPC specification.

In addition, the wsdl:service element is mapped to the implementation-specific ServiceLocator class, which is an implementation of the Generated Service interface.

Mapping between WSDL and SOAP messages

The WSDL file defines the format of the SOAP message that is sent over the wire. The **WSDL2Java** command and the WebSphere Application Server run time use the information in the WSDL file to confirm that the SOAP message is properly serialized and deserialized.

Document versus RPC, literal versus encoded

If a wsdl:binding indicates a message is sent using an RPC format, the SOAP message contains an element defining the operation. If a wsdl:binding indicates the message is sent using a document format, the SOAP message does not contain the operation element.

If the wsdl:part is defined using the type attribute, the name and type of the part are used in the message. If the wsdl:part is defined using the element attribute, the name and type of the element are used in the message. The element attribute is not allowed by the JAX-RPC specification when use="encoded".

If a wsdl:binding indicates a message is encoded, the values in the message are sent with xsi:type information. If a wsdl:binding indicates that a message is literal, the values in the message are typically not sent with xsi:type information. For example:

```
WSDL:

<xsd:element name="c" type="xsd:int" />
  ...
 <wsdl:message name="request">
  <part name="a" type="xsd:string" />
  <part name="b" element="ns:c" />
 </wsdl:message>
 ...
 <wsdl:operation name="method" >
  <input message="request" />
 ...
RPC/ENCODED:
<soap:body>
  <ns:method>
   <a xsi:type="xsd:string">ABC</a>
   <element attribute is not allowed in rpc/encoded mode>
  </ns:method>
 </soap:body>
DOCUMENT/LITERAL:
<soap:body>
 <a>ABC</a>
 <c>123</a>
</soap:body>
```

```
DOCUMENT/LITERAL  wrapped:
<soap:body>
 <ns:method_wrapper>
  <a>ABC</a>
  <c>123</a>
 <ns:method_wrapper>
</soap:body>
```

The document and literal wrapped mode is the same as the document and literal mode. However, in the document and literal wrapped mode, there is only a single element within the body, and the element has the same name as the operation.

Multi-ref processing

If `use=encoded`, XML types that are not simpleTypes are passed in the SOAP message using the `multi-ref` attributes, `href` and `id`. The following example assumes that parameters one and two reference the same Java bean named, `info` containing fields `a` and `b`:

Note:

Deserialization produces a single instance of the `info` class for the encoded case and two instances are created for the literal case.

```
RPC/ENCODED:
<soap:body>
 <ns:method>
  <param1 href="#id1" />
  <param2 href="#id2" />
 <ns:method>
 <multiref id="id1" xsi:type="ns:info">
  <a xsi:type="xsi:string">hello<a>
  <b xsi:type="xsi:string">world</b>
 </multiref>
</soap:body>
```

```
RPC/LITERAL:
<soap:body>
 <ns:method>
  <param1>
   <a>hello</a>
   <b>world</b>
  </param1>
  <param2>
   <a>hello</a>
   <b>world</b>
  </param2>
 <ns:method>
</soap:body>
```

XML arrays and the `maxOccurs` attribute

A SOAP message is affected by whether the element is defined by an XML array or using the `maxOccurs` attribute.

```
WSDL:
<element name="foo" type="ns:ArrayOfString" />
```

```
Literal Instance:

 <foo>
  <item>A</item>
   <item>B</item>
  <item>C</item>
 </foo>
```

```
WSDL:
<element name="foo" maxOccurs="unbounded" type="xsd:string"/>

Literal Instance:

 <foo>A</foo>
  <foo>B</foo>
 <foo>C</foo>
```

minOccurs and nillable attributes

An element specified with minOccurs=0 that has a null value is not serialized in the SOAP message. An element specifying nillable="true" has a null value and is serialized into a SOAP message with the xsi:nil=true attribute. For example:

```
<a xsi:nil="true" />
```

Qualified versus unqualified

The XML schema attributeForm and elementForm attributes indicate whether the attributes and nested elements are serialized with qualified or unqualified names. If a part name is serialized, it is always serialized as an unqualified name.

Installing IBM Web Services Development Kit for z/OS

The IBM Web Services Developer Kit for z/OS contains the following command-line tools that are used in developing and implementing Web services:

- **Java2WSDL**
- **WSDL2Java**
- **endptEnabler**

The IBM Web Services Developer Kit for z/OS is included with the WebSphere Application Server distributed product and located in the following directory:

```
<WAS_HOME>/webservices/bin/waszos_dk.exe
```

To install the IBM Web Services Developer Kit for z/OS:

1. Install and Customize your WebSphere Application Server.
2. Locate the IBM Web Services Developer Kit for z/OS executable file in the <WAS_HOME>/webservices/bin/ directory.
3. Right-click **waszos_dk.exe** > **Open** to start the installation.
4. Select the setup language and click **OK**.
5. Follow the InstallWizard prompts to complete the installation.

Develop a Web service.

Java2WSDL command

The IBM Web Services Developer Kit for z/OS contains the **Java2WSDL** command-line tool needed for developing and implementing Web services. See Installing the IBM Web Services Developer Kit for z/OS to start using the tool.

The **Java2WSDL** command maps a Java class to a Web Services Description Language (WSDL) file by following the Java API for XML-based remote procedure call (JAX-RPC) specification. The **Java2WSDL** command accepts a Java class as input and produces a WSDL file representing the input class. If there is an existing file at the output location, it is overwritten. The WSDL file generated by the

Java2WSDL command contains WSDL and XML schema constructs that are automatically derived from the input class. You can override these default values with command-line arguments.

Command line syntax and arguments

The command line syntax is:

```
Java2WSDL [argument...] class
```

The following command-line arguments are supported:

Required arguments

- *class*

  Represents the fully qualified name of one of the following Java classes:
  - Stateless session EJB remote interface that extends the javax.ejb.EJBObject class
  - Service Endpoint Interface that extends the java.rmi.Remote class
  - Java bean

  The **Java2WSDL** command locates the class in CLASSPATH.

Important arguments

- **-bindingName** *name*

  Specifies the name to use for the binding element. If not specified, the binding name is the portTypeName.
- **-help**

  Displays the help message.
- **-helpX**

  Displays the help message for extended options.
- **HelpXoptions**
  - **-debug**

    Displays debug messages.
  - **-outputImpl** *impl-wsdl*

    Specifies if you want an interface and implementation WSDL file emitted.
  - **-locationImport** *location-uri*

    Specifies the location of the interface WSDL file if you use the -outputImpl argument specified.
  - **-MIMEStyle**

    Specifies a style representing Multipurpose Internet Mail Extensions (MIME) information. Valid arguments are:
    - **Axis**
    - **WSDL11** (default)
  - **-soapAction**

    Valid arguments are:
    - **DEFAULT**

      Sets the `soapAction` field according to deployment information.
    - **NONE**

      Sets the `soapAction` field to "".
    - **OPERATION**

      Sets the `soapAction` field to the operation name.
  - **-stopClasses** *parent* **[,** *parent***]**

    If the -all argument is specified, the **Java2WSDL** command searches inherited classes and interfaces to construct the list of methods for WSDL file

operations. The **Java2WSDL** command searches inherited classes and interfaces when generating extended complexTypes. The search stops when a class or interface is found within a package that begins with `java` or `javax`. The -stopClasses argument can be used to define additional classes that cause the search to stop.

– **-namespaceImpl** *namespace*

  Specifies the target namespace for the implementation WSDL if -outputImpl specified.

– **-voidReturn**

  Valid arguments are:

  - **ONEWAY**

    Methods with void returns are one-way. This is the default for JMS transport.

  - **TWOWAY**

    Methods with void returns are two-way. This the default for HTTP transport.

– **-wrapped** *boolean*

  Specifies if the WSDL file should be generated according to wrapped rules. This is only valid if `use` is literal. The option defaults to `true`.

- **-extraClasses** *classes*

  Specifies other classes that should be represented in the WSDL file.

- **-input** *wsdl-uri*

  Specifies the input WSDL file used to build an output WSDL file. Information from an existing WSDL file, whose name is specified in this option, is used with the input Java class to generate the desired output.

- **-implClass** *impl-class*

  The **Java2WSDL** command uses method parameter names to construct the WSDL file message part names. The command automatically obtains the message names from the debug information in the class. If the class is compiled without debug information, or if the class is an interface, the method parameter names are not available. In this case, you can use the -implClass argument to provide an alternative class from which to obtain method parameter names. The impl-class does not need to implement the class if the class is an interface, but it must implement the same methods as class.

- **-location** *location*

  Provides the location or Uniform Resource Locator (URL) of the service. Typically, this value fills automatically when the Web service deploys. Use this argument to specify the location if you want to generate a WSDL file containing a location URL without deploying. A warning displays to remind you that the generated WSDL file should not be published if the final location is not yet been determined. The name after the last slash or backslash is the name of the service port, unless the name is overridden by the -servicePortName argument. The service port address location attribute is assigned the specified value.

- **-namespace** *targetNamespace*

  Indicates the target namespace for the WSDL file being generated. See Mapping between Java, WSDL and XML for the algorithm used to obtain the default namespace.

- **-output** *wsdl-uri*

  Indicates the path and file name of the output WSDL file. If not specified, the default file, `*class*.wsdl`, is written into the current directory.

- **-PkgtoNS** *package namespace*

Specifies the mapping of a Java package to a namespace. If there is a package without a namespace, the **Java2WSDL** command generates a namespace name. This argument can be repeated to specify mappings for multiple packages.

- **-portTypeName** ***name***

  Specifies the name to use for the portType element. If not specified, the class name is used.
- **-serviceElementName** ***name***

  Specifies the name of the service element.
- **-servicePortName** ***name***

  Specifies the name of the service. If not specified, the service name is derived from the -location argument.
- **-style RPC | DOCUMENT**

  Specifies the WSDL style to use in the generated WSDL file. For more information about styles, see Mapping between Java, WSDL and XML. This argument is used with the -use argument.

  If RPC is specified with -use ENCODED, or omitting use, a `style=rpc/use=encoded` WSDL file is generated. If RPC is specified with -use LITERAL, a `style=rpc/use=literal` WSDL file is generated. If DOCUMENT is specified with -use LITERAL or omitting use, a `style=document/use=literal` WSDL file is generated.
- **-transport http | jms**

  Generates SOAP bindings for either Hyper Text Transport Protocol (HTTP) (default) or Java Messaging Service (JMS). If `jms` is specified, the characters "jms" are appended to the WSDL file name to prevent overwriting an existing WSDL file for another transport. The transport option can only be specified once.
- **-use LITERAL | ENCODED**

  Specifies which style and use combinations are generated into the WSDL file when used with the -style argument. The combinations are rpc and encoded, rpc and literal, or doc and literal. For more information, see the Mapping between Java language, WSDL and XML.
- **-verbose**

  Displays verbose messages.

WSDL2Java command

The IBM Web Services Developer Kit for z/OS contains the **WSDL2Java** command-line tool needed for developing and implementing Web services. See Installing the IBM Web Services Developer Kit for z/OS to start using the tool.

The **WSDL2Java** command tool creates Java classes and deployment descriptor templates from a Web Services Description Language (WSDL) file using the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification. See Mapping between Java language, WSDL and XML for more information.

Classes and files generated

The following kinds of classes and files are generated:

- **For each portType in the WSDL document (`<wsdl:portType>` element tag):**
  - Service Endpoint Interface
- **For each service in the WSDL document (`<wsdl:service>` element tag):**
  - Service Interface when the -role develop-client argument is specified.
  - ServiceLocator when the -role deploy-client argument is specified.

    This class is a WebSphere product-specific implementation of the service interface, and is not used directly.

- webservices.xml deployment descriptor template when the -role develop-server argument is specified
  - ibm-webservices-bnd.xmi deployment descriptor template when the -role develop-server argument is specified.
  - ibm-webservices-ext.xmi deployment descriptor template when the -role develop-server argument is specified.
  - *wsdlfile*_mapping.xml JAX-RPC mapping file when the -role develop-client or -role develop-server is specified.
  - webservicesclient.xml deployment descriptor template when the -role develop-client argument is specified.
  - ibm-webservicesclient-bnd.xmi deployment descriptor template when the -role develop-client argument is specified.
  - ibm-webservicesclient-ext.xmi deployment descriptor template when the -role develop-client argument is specified.

  When the role is a server role, the container argument specifies which J2EE container the implementation uses. When the -role develop-server -container ejb arguments are specified, the webservices.xml, ibm-webservices-bnd.xmi, ibm-webservicesclient-ext.xmi and the mapping file are generated into the META-INF subdirectory. When the -role develop-server -container web arguments are specified, the files are generated into the WEB-INF directory.
- **For each binding in the WSDL file (<wsdl:binding> element tag):**
  - A stub that implements the Service Endpoint Interface (deploy-client role)
  - An implementation template for an enterprise bean and templates for the EJB remote interface and home interface generate when the -role develop-server and -container-ejb arguments are specified.
  - An implementation template for the Java bean when the -role develop-server and -container-web arguments are specified.
- **Other classes and files:**
  - A Java bean representing the structure of the type when the -role develop-server or -role develop-client arguments are specified for each complexType or simpleType.
  - Three classes, *_Ser.java, *_Deser.java, and *_Helper.java, generate for each complexType to assist in converting the bean to SOAP and back when the -role deploy-server or -role deploy-client argument is specified.
  - A *Holder.java class generates when the -role develop-server or -role develop-client arguments are specified for each out and inout parameter.

Command line syntax

The command line syntax is:

```
WSDL2Java [arguments] WSDL-URI
```

Required arguments

- ***WSDL-URI***

  Specifies the location of the input WSDL document using a Universal Resource Identifier (URI). You can also use a regular file path if the WSDL file is on the local file system.

Important arguments

- **-container** *j2ee-container*

  Indicates the J2EE container to be used. Valid arguments are:
  - **client**

    Indicates client container.
  - **ejb**

Indicates enterprise JavaBean (EJB) container.
- **none**

  Indicates no container.
- **web**

  Indicates Web container.

If client is role, the default argument is **none**. If server is role, the container must be **ejb** or **web**. The same container option must be used for both development and deployment.

- **-deployScope** *argument*

  Indicates how to deploy the server implementation. Valid arguments are:
  - **Application**

    Uses one instance of the implementation class for all requests.
  - **Request**

    Creates a new instance of the implementation class for each request.
  - **Session**

    Creates a new instance of the implementation class for each session.
- **-genResolver**

  Generates an `absolute-import resolver` class. The purpose of this class is to record the contents of the imported WSDL files used by the WSDL URI. This class is used by the runtime. It can also be used for future **WSDL2Java** command runs. This is desirable when the imported WSDL files are remote and can be inaccessible or slow to access. It also eliminates the possibility that a remote WSDL file might have different contents at run time than it did at development time. The generated class is named `_AbsoluteImportResolver.java`. You should compile and package this class with the other Java classes generated by the **WSDL2Java** command.
- **-help**

  Displays a help message and exits.
- **-helpX**

  Displays a help message for extended options and exits. The options are:
  - **-all**

    Generates Java files for all types, even those that are not referenced.
  - **-debug**

    Prints debugging information.
  - **-fileNStoPkg** *filename*

    Specifies the file of namespace to package mappings. The default is `NStoPKG.properties`.
  - **-genJava** *argument*

    Generates Java files. Valid arguments are:
    - **IfNotExists**, default
    - **Overwrite**
    - **No**
  - **-genXML** *argument*

    Generates the `.xml` and `.xmi` files. Valid arguments are:
    - **IfNotExists**, default
    - **Overwrite**
    - **No**
  - **-password** *password*

    Specifies the login user password to access the WSDL URI.
  - **-testCase**

    Generates the template for a JUnit test case for testing a Web service.

– **-user** *id*

  Specifies the login user name to access the WSDL URI.

- **-inputMappingFile** mapping file

  Specifies the file name of the Java to WSDL mapping file.

- **-NStoPkg** *namespace=package*

  By default, package names are automatically derived from the namespace strings in the WSDL file. For example, if the namespace is of the form *http://x.y.com* or *urn:x.y.com*, the corresponding package is *com.y.x*.

  You can provide your own mapping by using the **-NStoPkg** argument, which you can repeat as often as necessary, once for each unique namespace mapping. For example, if there is a namespace in the WSDL file called `urn:AddressFetcher2`, and you want files generated from the objects in this namespace to reside in the package `samples.addr`, provide the -NStoPkg urn:AddressFetcher2=samples.addr argument to the **WSDL2Java** command.

- **-output** directory

  Sets the root directory for emitted files.

- **-role** *j2ee role*

  Specifies the J2EE development role that identifies which files to generate. Valid arguments are:

  – **client**

    Combination of develop-client and deploy-client.

  – **deploy-client**

    Generates binding files for client deployment.

  – **deploy-server**

    Generates binding files for server deployment.

  – **develop-client** (default)

    Generates files for client development.

  – **develop-server**

    Generates files for server development.

  – **server**

    Combination of develop-server and deploy-server.

- **-timeout** *seconds*

  Specifies how long the **WSDL2Java**command should wait, in seconds, for the WSDL-URI to respond before giving up. The default is 45 seconds, -1 disables the timeout.

- **-useResolver** *resolver-class*

  Specifies an `absolute-import resolver` class to use during parsing. This class must have been created during a previous execution of the **WSDL2Java** command using the -genResolver option. The class must be available in CLASSPATH.

- **-verbose**

  Displays processing information, including the names of the generated files.

Setting up a development and unmanaged client execution environment for Web services based on Web Services for J2EE

WebSphere Application Server provides command-line tools to develop Web services clients and implementations that are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification. WebSphere Application Server also includes the Assembly Toolkit that can be downloaded from the Web site

http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

Websphere Studio Application Developer Version 5.1 has GUI-based development tools to develop Web services that integrate with Websphere Application Server 5.0.2.

Before you can set up a Web services development and unmanaged client execution environment within WebSphere Application Server, you must Install WebSphere Application Server.

To set up a Web services development and unmanaged client execution environment:

1. Develop application code and run the setupCmdLine script.
2. Configure the path. You can add the WebSphere and Java `bin` directories to your path by typing:

   On Windows platforms:

   ```
   set PATH=%WAS_PATH%;%PATH%
   ```

   On UNIX:

   ```
   export PATH=$WAS_PATH:$PATH
   ```

Develop Web services based on Web Services for J2EE.

Developing a Web service from a Java bean

Set up a Web services development and unmanaged client execution environment.

To develop a Web service from a Java bean:

1. Access an existing Java bean Web archive (WAR) file.
2. Develop a Java bean Service Endpoint Interface.
3. Develop a Web Services Description Language (WSDL) file.
4. Develop Web services deployment descriptor templates for a Java bean implementation.
5. Configure the `webservices.xml` deployment descriptor.
6. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
7. Assemble a Web services-enabled WAR file when starting from Java.
8. Assemble a Web services-enabled WAR into an EAR file.
9. Deploy the EAR file into WebSphere Application Server.

Test the Web service.

Developing a WSDL file

Develop a Service Endpoint Interface.

You need a Web Services Description Language (WSDL) file to use Web services. You can develop your own WSDL file or get one from a Web service provider through E-mail, downloading or through a Uniform Resource Locator (URL). This documentation assumes you are creating your own.

To develop a WSDL file:

1. Configure the Service Endpoint Interface class and referenced classes into your CLASSPATH.
   - On Windows, set CLASSPATH="%CLASSPATH%;*<list your application JAR files and classes>*".
   - On UNIX, export CLASSPATH="$CLASSPATH:*<list your application JAR files and classes>*".
2. Run the **Java2WSDL *seiInterface*** command. A WSDL file named *seiInterface*.wsdl is created.
   - Move the WSDL file to the META-INF/wsdl subdirectory if you are using an enterprise JavaBean (EJB).
   - Move the WSDL file to the WEB-INF/wsdl subdirectory if you are using a Java bean.
3. Edit the generated WSDL file and inspect the part names. The WSDL parts have names like arg_0_0. Modify the WSDL file to use the actual names of the Java parameters.
4. (Optional) Use the **Java2WSDL** command tool to generate the correct part names of WSDL file. You can automatically generate and set the correct part names by using the **Java2WSDL** command tool. Generating and setting the part names is done by providing additional information to the **Java2WSDL** command tool in the form of a Java implementation class that implements the same methods as the Service Endpoint Interface and is compiled with debug information on (**javac -g**). Parameter names are stored in the .class file with the debug information. If your implementation class was compiled with debug on, you can use the **Java2WSDL -implClass *seiImpl seiInterface*** command to generate a WSDL file having the proper part names.

A WSDL file that defines the Web service described by the Service Endpoint Interface.

This example uses a JAR file named AddressBook.jar containing a class named AddressBook.class file.

You must add the AddressBook.jar file to your CLASSPATH to create the WSDL file. The JAR file contains an EJB implementation class that was compiled with debugging information on. Run the **Java2WSDL -implClass addr.AddressBookBean addr.AddressBook** command to create a WSDL file named AddressBook.wsdl.

Develop Web services deployment descriptor templates from a WSDL file.

WSDL:

Web Services Description Language (WSDL) is an eXtensible Markup Language (XML)-based description language that has been submitted to the W3C as the industry standard for describing Web services. The power of WSDL is derived from two main architectural principles: the ability to describe a set of business operations and the ability to separate the description into two basic units, a description of the operations and the details of how the operation and the information associated with it are packaged.

A WSDL document allows a service provider to specify the name and address of the Web service; protocol and encoding style used when accessing the public

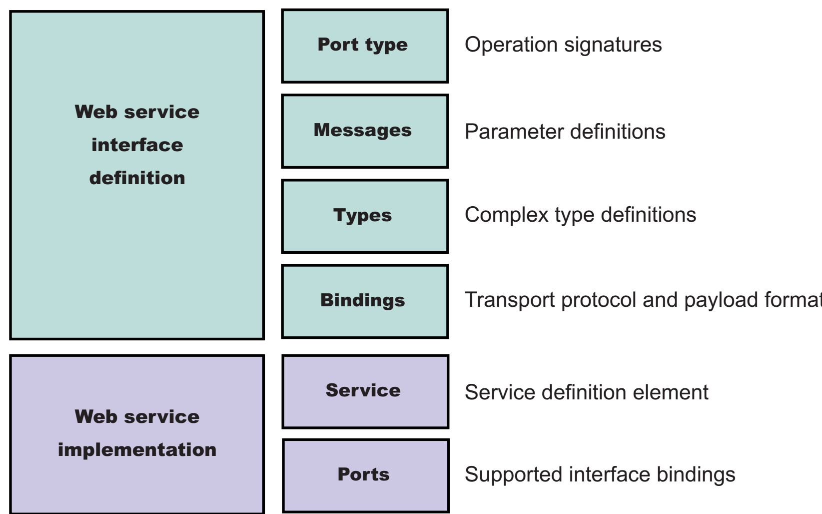
operations of the Web service; and the type information, including name, operations, parameters and data comprising the interface of the Web service.

The WSDL document is the engine of a Java 2 platform, Enterprise Edition (J2EE) Web service; without it there is no service. The information within a WSDL file maps to the Java application to create a Web service.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Therefore, a WSDL document is composed of several elements. See WSDL anatomy for more information and examples of the WSDL elements.

When creating a Web service for WebSphere Application Server, you must first have an implementation bean that includes a Service Endpoint Interface. Then, you use the **Java2WSDL** command-line tool to create a WSDL that defines the Web service. To learn more about how the WSDL file is used in the development process, see Developing Web services based on Web Services for J2EE.

WSDL anatomy: Web Services Description Language (WSDL) files are written in eXtensible Markup Language (XML). To learn more about XML, see Web services: Resources for learning.



A WSDL contains the following parts:

- **Web service interface definition**

  This is where the elements are contained, as well as the namespaces.
- **Web service implementation**

  This is where you find the definition of the service and ports.

A WSDL file describes a Web service with the following elements:

portType

The description of the operations and their associated messages. PortTypes define abstract operations.

```
<portType name="EightBall">
 <operation name="getAnswer">
  <input message="ebs:IngetAnswerRequest"/>
  <output message="ebs:OutgetAnswerResponse"/>
 </operation>
</portType>
```

message

The description of parameters (input and output) and return values.

```
<message name="IngetAnswerRequest">
 <part name="meth1_inType" type="ebs:questionType"/>
</message>
<message name="OutgetAnswerResponse">
 <part name="meth1_outType" type="ebs:answerType"/>
</message>
```

types

The schema for describing XML complex types used in the messages.

```
<types>
 <xsd:schema targetNamespace="...">
  <xsd:complexType name="questionType">
   <xsd:element name="question" type="string"/>
  </xsd:complexType>
  <xsd:complexType name="answerType">
  ...
</types>
```

binding

Bindings describe the protocol used to access a service, as well as the data formats for the messages defined by a particular portType.

```
<binding name="EightBallBinding" type="ebs:EightBall">
 <soap:binding style="rpc" transport="schemas.xmlsoap.org/soap/http">
 <operation name="ebs:getAnswer">
 <soap:operation soapAction="urn:EightBall"/>
  <input>
   <soap:body namespace="urn:EightBall" ... />
  ...
```

The remaining parts, services and ports, indicate where you can find the WSDL.

Service

Contains the Web service name and a list of the ports.

Ports

Contains the location of the Web service and the binding to used to access the service.

```
<service name="EightBall">
 <port binding="ebs:EightBallBinding" name="EightBallPort">
  <soap:address location="localhost:8080/axis/EightBall"/>
 </port>
</service>
```

Publishing WSDL files:

To publish a Web Services Description Language (WSDL) file you need an enterprise application, also known as an enterprise archive (EAR) file, that contains a Web services-enabled module and has been deployed into WebSphere Application Server. See Deploying Web services based on Web Services for Java 2 platform, Enterprise Edition (J2EE).

The WSDL files for each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to clients that want to invoke your Web services.

You can publish WSDL files for the deployed EAR file in one of three ways:
1. Publish a WSDL file with the administrative console.
2. Publish a WSDL file with the **wsadmin** command tool.
3. Publish a WSDL file through a URL.

Publishing WSDL files with the administrative console:

When publishing Web Services Description Language (WSDL) files with the administrative console, you can specify default or custom HTTP URL prefixes. You can also specify a Java Message Service (JMS) URL prefix.

To publish a WSDL file with the administrative console:
1. Open the administrative console.
2. Click **Applications**> **Enterprise Applications** > *application*. Under Additional Properties, click **Publish WSDL** which brings you to the **Publish WSDL files for Web Services** panel.
3. Specify the default URL prefixes for the Web service.
   a. Select **HTTP URL prefix**.
   b. Select an entry from the drop down list. If you have multiple application modules, select the application module's checkbox on the module table.
   c. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   d. Click **OK**.
   e. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.
   f. Download the zip file. Follow your browser's instructions to download the zip file.
4. Specify custom URL prefixes for the Web service.
   a. Select **Custom HTTP URL prefix**.
   b. Type the name of the URL prefix in the **Custom HTTP URL prefix** field. The entry must be of the form http|https://<*host_name*>:<*port_number*>. For example:
      ```
      http://myHost:999
      ```
      If you have multiple application modules, select the application module's checkbox on the module table.
   c. Click **Apply**. The URL prefix is copied to the selected module HTTP URL prefix field.
   d. Click **OK**.
   e. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.

f. Download the zip file. Follow your browser's instructions to download the zip file.

5. Specify a JMS URL prefix.
   a. Select the application module.
   b. Type the JMS URL prefix into the **JMS URL prefix** field. The entry must be of the form: jms:/[*queue*|*topic*]?destination=<*queue* or *topic_jndi_name*>&connectionFactory=<*connection_factory_jndi_name*>. For example:

      jms:/*queue*?destination=jms/Q1&connectionFactory=jms/QCF1
   c. Click **OK**.
   d. Click the exported *WSDL_zip_file* listed on the **Export WSDL Zip file** panel.
   e. Download the zip file. Follow your browser's instructions to download the zip file.

Publishing WSDL files using the wsadmin command:

The Web Services Description Language (WSDL) files in each Web services-enabled module are published to the file system location you specify. You can provide these WSDL files to the clients that want to invoke your Web services.

The scripting client, **wsadmin**, can publish the WSDL files in either local, for example, -conntype NONE, or remote mode. However, in local mode, the target application should be located at the same node where the **wsadmin** command is invoked.

The steps below assume that the application has been deployed and that the application server is running.

To publish a WSDL file with the **wsadmin** command:

1. From a command prompt, start *install_root*\bin\wsadmin if you are using Windows or *install_root*/bin/wsadmin if you are using UNIX.
2. At the **wsadmin** command prompt, enter one of the two commands:
   - **$AdminApp publishWSDL** ***app_Name path_Name***
   - **$AdminApp publishWSDL** ***app_Name path_Name*** **{{module {{*binding url-prefix*}}}}**

   Where
   - *app_Name* is the application name
   - *path_Name* is the absolute path to the zip file that will contain the published WSDL files. The zip file is saved on the machine running WebSphere Application Server, therefore, if the server is running on a different machine, you need to obtain the zip file from that machine. The directory structure of the resulting zip file is based on the following information:

     ```
     Application file name
             module file name
                 META-INF/ or WEB-INF/
                     wsdl/
                           WSDL file name
     ```

     See the usage scenario for an example of this directory structure.
   - *binding* is either http or jms (both are in lower case)
   - *url-prefix* is the partial SOAP address for the associated SOAP binding. For an HTTP binding the form is `http://host:port/` or `https://host:port`.

For Java Message Service (JMS) bindings, the form is `jms:/queue?destination=`*`dest`*`&connectionFactory=cf` or `jms:/topic?destination=`*`dest`*`&connectionFactory=cf`

The **$AdminApp publishWSDL** *app_Name path_Name* command updates the WSDL SOAP address prefixes using the default values. If you do not want to update the WSDL SOAP address prefixes, use the other command, instead of the default values.

The **$AdminApp publishWSDL** *app_Name path_Name* **{{module {{*binding url-prefix*}}}}** command allows you to customize the WSDL SOAP address for each module. You can specify a different address prefix for each SOAP binding.

The WSDL files from Web services are published to a specified zip file. You can hand the zip file to the client and the client can use the published WSDL files to create a Web services client that accesses the deployed service.

The command to publish WSDL files for a Web service named WebServicesSamples could be **$AdminApp publishWSDL WebServicesSamples c:/temp/samplesWsdl.zip**

or

$AdminApp publishWSDL WebServicesSamples c:/temp/sampleswsdl.zip { {AddressBookJ2WB.war {{http http://localhost:9080}}} {StockQuote.jar {{http https://localhost:9443}}} }

The directory structure for this created zip files is

```
WebServicesSamples.ear/StockQuote.jar/META-INF/wsdl/StockQuoteFetcher.wsdl
WebServicesSamples.ear/AddressBookW2JE.jar/META-INF/wsdl/AddressBookW2JE.wsdl
WebServicesSamples.ear/AddressBookJ2WE.jar/META-INF/wsdl/AddressBookJ2WE.wsdl
WebServicesSamples.ear/AddressBookJ2WB.war/WEB-INF/wsdl/AddressBookJ2WB.wsdl
WebServicesSamples.ear/AddressBookW2JB.war/WEB-INF/wsdl/AddressBookW2JB.wsdl
```

Publishing WSDL files using a URL:

Before you can publish a Web Services Description Language (WSDL) file using a URL, the Web services-enabled application should be installed and running.

The files referenced by the `<wsdl-file>` element in the `webservices.xml` file can or cannot import other WSDL or XSD files. Typically, all WSDL or XSD files are originally placed into the `META-INF/wsdl` directory when using enterprise JavaBeans (EJBs) or the `WEB-INF/wsdl` directory when using Java beans. If your WSDL or XSD files are not placed in one of these directories, the file referenced by the `<wsdl-file>` and its imported files are located at the same directory and copied to the `wsdl/` directory for publishing purposes.

Note: EJB-based Web service applications must have an HTTP router or a Web module. Only HTTP URLs are supported for publishing.

To publish a WSDL file using a URL:

1. Retrieve the outer-most WSDL file. The outer-most WSDL file is the WSDL file defined by the `<wsdl-file>` element in the `webservices.xml` file.

   Each Web service has an endpoint address, like `http://example.com/services/stockquote`. You can retrieve the outer-most WSDL file (defined by the <wsdl-file> element within the webservices.xml file)

by appending the string "/wsdl" or "/wsdl/" to the endpoint address, for example,`http://example.com/services/stockquote/wsdl`.

2. Retrieve the imported WSDL files. When the outer-most WSDL file imports other WSDL or XSD files, these imported files can be retrieved by appending the relative path to the URL, which is used to retrieve the outer-most WSDL file. This is also true for WSDL files that import other files. This process is similar to the use of relative hyperlinks in HTML documents. If an HTML document contains a hyperlink to other documents, the relative path is appended to create the URL to access the hyperlinked documents.

Suppose you have an application with the following directory structure:

```
<module-root>/
META-INF/
WEB-INF/
webservices.xml/* define Foo service, the <wsdl-file> element points to "/wsdl/fooImpl.wsdl" */
web.xml
ibm-webservices-bnd.xml
<jaxrpc-mapping-file>
wsdl/
fooImpl.wsdl/* importing foo.wsdl which is an interface wsdl */
foo.wsdl /* importing type definition for the interface */
fooTypes.xsd
```

If the SOAP address for the foo service is `http://examples.com:9080/services/foo`, the simple way to retrieve the foo's outer-most WSDL, is with the following form `http://examples.com:9090/services/foo/wsdl` or `http://examples.com:9090/services/foo/wsdl/`. The URL is redirected to `http://examples.com:9090/services/foo/wsdl/fooImpl.wsdl`, where `fooImpl.wsdl` is the name of the outer-most WSDL file.

Since the `fooImpl.wsdl` file has the import `<import namespace="http://examples.com/foo" location="a/b/foo.wsdl>`, use the URL `http://examples.com:9090/services/foo/wsdl/a/b/foo.wsdl` to obtain the `foo.wsdl` file.

Publish WSDL files settings:

Use this page to publish Web Services Description Language (WSDL) files.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Publish WSDL**.

When you click **OK**, a zip file of all the Web services-enabled modules in the application is produced. The name of the published zip file is `application_name_WSDLFiles.zip`. In the published zip file, the directory structure is *application_name/module_name/[META-INF|WEB-INF]/wsdl/wsdl_file_name*.

In a published WSDL file, the location attribute of a service `soap:address` stanza contains the URL through which the Web service is accessed. You can specify the portion of the URL to be used for the Web services in each module. You can access the Web services in a module through a HTTP transport or JMS transport, or both. You can specify URL information for both types of transports.

Specify URL prefixes for Web Services:

Specifies the *protocol* (either http or https), *host_name*, and *port_number* to be used in the URL.

The URL prefix format is protocol://*host_name:port_number*, for example, http://*myHost*:9045. The actual URL that appears in a published WSDL file consists of the prefix prepended to the module's context-root and the Web service url-pattern, for example, http://*myHost*:9045/services/*myService* .

Select HTTP URL prefix:

Specifies the drop down list associated with a default list of URL prefixes. This list is the intersection of the set of ports for the module's virtual host and the set of ports for the module's application server. Use items from this list if the Web services application server is accessed directly.

To set an HTTP prefix, select either the **HTTP URL prefix** or **Custom HTTP URL prefix**, enter the value, select the check box of the modules that are to use the prefix, and click **Apply**. When you click **Apply**, the entry in the **Select HTTP URL prefix** or **Custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP URL prefix** field of any module whose check box (in the leftmost column) is selected. The HTTP prefix is not applied to the fields in the JMS URL prefix column.

Custom HTTP URL prefix:

Specifies the *protocol*, *host*, and *port_number* of the intermediate service if the Web services in a module are accessed through an intermediate node, for example the Web services gateway or an IHS server.

To set an HTTP prefix, select either the **HTTP URL prefix** or **Custom HTTP URL prefix**, enter the value, select the check box of the modules that are to use the prefix, and click **Apply**. When you click **Apply**, the entry in the **Select HTTP URL prefix** or **Custom HTTP URL prefix** fields, depending on which is selected, is copied into the **HTTP URL prefix** field of any module whose check box (in the leftmost column) is selected. The HTTP prefix is not applied to the fields in the JMS URL prefix column.

JMS URL prefix:

Specifies the JMS URL prefix string used for each module.

The URL prefix specified must contain the destination and connectionFactory properties. It can contain other property-value pairs, but it must not contain the targetService property, which is added by the system when the published WSDL files are created. The format of the JMS URL prefix is jms:/[*queue&topic*]?destination =*target_queue_or_topic_jndi_name*&connectionFactory=*factory_jndi_name*, for example, jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF. The actual URL that appears in a published WSDL file consists of the prefix prepended to the Web service targetService, for example, jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF&targetService=StockQuote.

Multipart WSDL best practices: WebSphere Application Server supports deployment of Web services using a multipart Web Services Description Language (WSDL) file. That is, WSDL files import other WSDL files when the WSDL file listed in the `<wsdl-file>` element of the `webservices.xml` deployment descriptor

contains all `<wsdl:service>` and `<wsdl:port>` elements. The WSDL file is divided into an implementation WSDL and an interface WSDL.

The `<wsdl:import>` element indicates a reference to another WSDL file. If the `<wsdl:import>` element `location` attribute does not contain a URL, that is, it contains only a file name, and does not begin with `http://`, `https://` or `file://`, the imported file must be located in the same directory and must not contain a relative path component. For example, if `META-INF/wsdl/A_Impl.wsdl` is in your module and contains the import statement `<wsdl:import="A.wsdl" namespace="..."/>` , the file, `A.wsdl` must also be located in the module `META-INF/wsdl` directory.

It is recommended that all WSDL files be placed in either the `META-INF/wsdl` directory, if you are using enterprise JavaBeans (EJBs), or the `WEB-INF/wsdl` directory, if you are using Java beans, even if there are relative imports within the WSDL files. Otherwise, there are implications with the WSDL publication when you use a path like the following `<location="../interfaces/A_Interface.wsdl"namespace="..."/>`. Using a path like this fails because the presence of the relative path, regardless of whether the file is located at that path or not. If the location is a URL, it must be readable at both deployment and server startup.

WSDL publication

The files located in the `META-INF/wsdl` or `WEB-INF/wsdl` directory can be published through either a URL or file, including WSDL or XSD files. For example, if the file referenced in the `<wsdl:file>` element of the `webservices.xml` deployment descriptor is located in the `META-INF/wsdl` or `WEB-INF/wsdl` directory, it is publishable. If the files imported by the `<wsdl:file>` are located in the `wsdl/` directory or its subdirectory, they are publishable.

If the WSDL file referenced by the `<wsdl:file>` element is located in a directory other than `wsdl`, or its subdirectories, the file and its imported files, either WSDL or XSD files, which are in the same directory, are copied to the `wsdl` directory without modification when the application is installed. These types of files can also be published.

If the `<wsdl:file>` imports a file located in a different directory, the file is not copied to the `wsdl` directory and not available for publishing.

Developing a Service Endpoint Interface for a Java bean implementation

Set up a Web services development and unmanaged client execution environment.

The Service Endpoint Interface defines the methods for a particular Web service. The Java bean implementation must implement methods having the same signature as the methods on the Service Endpoint Interface. There are a number of restrictions on which types to use as parameters and results of Service Endpoint Interface methods. These restrictions are documented in the Java API for XML-based remote procedure call (JAX-RPC) specification, which is available through Web services: Resources for learning.

You can also create a Service Endpoint Interface by using the Assembly Toolkit, which is a component of the Application Assembly Toolkit. The steps are similar except the Assembly Toolkit automatically compiles the interface when you save it.

To develop a Service Endpoint Interface for a Java bean implementation:

1. Create a Java interface containing the methods to include in the Service Endpoint Interface. The interface should extend the `java.rmi.Remote` interface. Each method throws the exception, `java.rmi.RemoteException`. If you start with an existing Java interface, remove any methods that do not conform to JAX-RPC.
2. Compile the interface.

A Service Endpoint Interface which you can use to develop a Web service.

This example uses a Java interface called AddressBook. The following example depicts the AddressBook interface:

```
package addr;
public interface AddressBook {
    /**
     * Retrieve an entry from the AddressBook.
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name);
}
```

You use the AddressBook Java interface to create the Service Endpoint Interface:

1. Begin with the remote interface, `AddressBook.java`.
2. Make a copy of the remote interface named `AddressBook_SEI.java` and use it as a template for the Service Endpoint Interface.
3. Change the interface to extend the `java.rmi.Remote` interface.
4. Modify each method declaration to add a throws clause for `java.rmi.RemoteException`.
5. Compile the interface.

Use the Service Endpoint Interface to Develop a Web Services Description Language (WSDL) file.

Developing Web services deployment descriptor templates for a Java bean implementation

To develop the deployment descriptor templates from a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: *file:drive:\path\file_name.*wsdl. If you are using the UNIX platform, the URL looks like this: *file:/path/file_name.*wsdl. You can also specify local files using the absolute or relative file system path.

When the Web service implementation is a Java bean in a Web module, the `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices.ext.xmi` deployment descriptors and the Java API for XML-based remote procedure call (JAX-RPC) mapping file are generated in the WEB-INF subdirectory.

To develop deployment descriptor templates:
Run the **WSDL2Java -verbose -role develop-server -container web -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and

mapping file into the WEB-INF subdirectory. If the **-verbose** option is specified, a list of all generated files displays when the command runs.

Deployment descriptor templates that are required to implement or use a Web service.

The following example uses a WSDL file named AddressBookJ2WB.wsdl:

1. Generate the template files:
   - WSDL2Java -verbose -role develop-server -container web -genJava no AddressBookJ2WB.wsdl

   The deployment descriptor templates and mapping file are generated into the WEB-INF subdirectory as follows:

```
Parsing XML file: AddressBookJ2WB.wsdl
Generating: WEB-INF\webservices.xml
Generating: WEB-INF\ibm-webservices-bnd.xmi
Generating: WEB-INF\ibm-webservices-ext.xmi
Generating: WEB-INF\AddressBookJ2WB_mapping.xml
```

Developing a Web service using a stateless session enterprise bean

Set up a Web services development and unmanaged client execution environment.

To use an enterprise bean as the basis for a Web service implementation, follow these requirements:

- The enterprise bean must be a stateless session bean.
- Web service method parameters must be serializable and cannot be object references.
- Web service method parameters must be one of the supported Java API for XML-based remote procedure call (JAX-RPC) types.

These requirements are documented in the JAX-RPC specification available through Web services: Resources for learning.

Create the artifacts that enable the enterprise bean to be a Web service and assemble the artifacts into the enterprise application as follows:

1. Access an existing Java archive (JAR) file to be used as a Web service. Make sure that the enterprise bean meets the requirements.
2. Develop an EJB Service Endpoint Interface. The Service Endpoint Interface defines which enterprise bean methods should be made available as a Web service.
3. Develop a Web Services Description Language (WSDL) file.
4. Develop Web services deployment descriptor templates from an EJB implementation.
5. Assemble a Web services-enabled JAR file.
6. Configure the webservices.xml deployment descriptor.
7. Configure the ibm-webservices-bnd.xmi deployment descriptor.
8. Assemble a Web services-enabled enterprise archive (EAR) file.
9. Enable the EAR file. When the EAR file contains EJB modules, it must have the Web services endpoint Web archive (WAR) file added with the endptEnabler tool before it is deployed.
10. Deploy the EAR file into WebSphere Application Server.

A Web service from a stateless session enterprise bean.

Developing a Service Endpoint Interface from an EJB remote interface

Set up a Web services development and unmanaged client execution environment.

The Service Endpoint Interface defines the Web services methods. The enterprise JavaBean (EJB) that implements the Web service must implement methods having the same signature as the methods of the Service Endpoint Interface. There are a number of restrictions on which types to use as parameters and results of Service Endpoint Interface methods. These restrictions are documented in the Java API for XML-based remote procedure call (JAX-RPC) specification, which is available through Web services: Resources for learning.

The easiest method for creating the Service Endpoint Interface for an EJB Web service implementation is from the EJB remote interface.

You can also create a Service Endpoint Interface by using the Assembly Toolkit, which is a component of the Application Server Toolkit. The steps are similar except the Assembly Toolkit automatically compiles the interface when you save it.

To develop a Service Endpoint Interface:

1. Create a Java interface containing the methods to include in the Service Endpoint Interface. The interface should extend the `java.rmi.Remote` interface. Each method throws the exception, `java.rmi.RemoteException`. If you start with an existing Java interface, remove any methods that do not conform to JAX-RPC.
2. Compile the interface.

A Service Endpoint Interface which you can use to develop a Web service.

This example uses an EJB remote interface called AddressBook_RI.

```
package addr;
public interface AddressBook_RI extends javax.ejb.EJBObject {
    /**
     * Retrieve an entry from the AddressBook.
     *
     *@param name the name of the entry to look up.
     *@return the AddressBook entry matching name or null if none.
     *@throws java.rmi.RemoteException if communications failure.
     */
    public addr.Address getAddressFromName(java.lang.String name)
     throws java.rmi.RemoteException;
}
```

You use the `AddressBook_RI` remote interface to create the Service Endpoint Interface:

1. Begin with the remote interface, `AddressBook_RI.java`:
2. Make a copy of the remote interface named `AddressBook.java` and use it as a template for the Service Endpoint Interface.
3. Change the interface to extend the `java.rmi.Remote` interface, instead of the `javax.ejb.EJBObject` Service Endpoint Interface.
4. Compile the `AddressBook.java` Service Endpoint Interface.

Use the Service Endpoint Interface to Develop a WSDL file.

Developing Web services deployment descriptor templates for an EJB implementation

To develop the deployment descriptor templates from a Web Services Description Language (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: *file:drive:\path\file_name.*wsdl. If you are using the UNIX platform, the URL looks like this: *file:/path/file_name.*wsdl. You can also specify local files using the absolute or relative file system path.

When the Web service implementation is an enterprise Java bean (EJB) in an EJB module, the `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` deployment descriptors, and the Java API for XML-based remote procedure call (JAX-RPC) mapping file are generated in the `META-INF` subdirectory.

To develop deployment descriptor templates:
Run the **WSDL2Java -verbose -role develop-server -container ejb -genJava no *wsdlURL*** command to generate the server deployment descriptor templates and mapping file into the `META-INF` subdirectory. If the **-verbose** option is specified, a list of all generated files displays when the command runs.

Deployment descriptor templates that are required to implement a Web service.

The following example uses a WSDL file named `AddressBookJ2WE.wsdl`:

1. Generate the template files:
   - `WSDL2Java -verbose -role develop-server -container ejb -genJava no AddressBookJ2WE.wsdl`

   The deployment descriptor templates are generated into the `META-INF` subdirectory as follows:

   ```
   Parsing XML file: AddressBookJ2WE.wsdl
   Generating: META-INF\webservices.xml
   Generating: META-INF\ibm-webservices-bnd.xmi
   Generating: META-INF\ibm-webservices-ext.xmi
   Generating: META-INF\AddressBookJ2WE_mapping.xml
   ```

Completing the EJB implementation

Develop EJB implementation templates and bindings from a Web Services Description (WSDL) file.

To complete the EJB implementation:

1. Inspect the enterprise EJB remote interface template, *portType*`_RI.java`. If necessary, modify the template. *portType* is the name of the `<wsdl:portType>` element in the WSDL file.
2. Inspect the EJB home interface template, *portType*`Home.java`. If necessary, modify the template.
3. Edit the EJB implementation template, *binding*`Impl.java`. *binding* is the name of the `<wsdl:binding>` element in the WSDL file.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.
4. Compile all the Java classes.

5. Assemble an EJB Java archive (JAR) file. Assemble all the Java classes into an EJB JAR file using the typical EJB assembly tools. Include all of the classes generated from running the **WSDL2Java** command tool when developing implementation templates and bindings from a WSDL file.

An EJB JAR file containing an EJB and supporting classes created from a WSDL file.

Configure the webservices.xml deployment descriptor .

Configuring the webservices.xml deployment descriptor

Create an enterprise JavaBean (EJB) Java archive (JAR) file or Web archive (WAR) file containing `webservices.xml`:

- Assemble a Web services-enabled EJB JAR file when starting from Java code.
- Assemble a Web services-enabled EJB JAR file from WSDL.
- Assemble a Web services-enabled WAR file when starting from Java code.
- Assemble a Web services-enabled WAR file when starting from WSDL.

Do one of the following based on whether your implementation is an EJB JAR file or Web module WAR file:

- Develop Web services Java bean deployment descriptor templates from a WSDL file.
- Develop Web services EJB deployment descriptor templates from a WSDL file.

This topic explains how to configure the `webservices.xml` deployment descriptor with the Assembly Toolkit which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

To configure the `webservices.xml` deployment descriptor:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file or WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the `webservicesclient.xml` file in the **Project Navigator** pane.
6. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents appear.
7. Right-click the `webservices.xml` file.
8. Select **Open**. The **Web Services** editor opens.
9. Expand the **Web service descriptions** section.
   a. Select the service you want to configure.
10. Expand the **Web service description implementation details** section.
    a. Verify the **Web service description name** field is unique among all the Web service descriptions in the editor.
    b. Verify that the **WSDL file** field indicates there is an existing WSDL file in the module. This file, by convention, should be located in the

`META-INF/wsdl` directory for an enterprise bean JAR file and in the `WEB-INf/wsdl` directory for a WAR file.

c. Verify the **JAX-RPC mapping file** field indicates an existing mapping file within the module. This file, by convention, should be located in the `META-INF` directory for an enterprise bean JAR file and in the `WEB-INF` directory for a WAR file.

11. Expand the **Port components** section.
    a. Verify there are port component entries corresponding to the used WSDL ports in the **Port components** section.
12. Select a *port_component* to open the editor for that port component. The **Port Components** editor opens.
13. Expand the **Port component implementation details** section.
    a. Verify the **WSDL Port Namespace URL** and **WSDL Port Local part** fields are set to the namespace and local name of the corresponding port in the WSDL file. These fields are configured by the **WSDL2Java** command tool when the `webservices.xml` file is generated.
14. Verify the **Service endpoint interface** field names the fully qualified Service Endpoint Interface class. This field is configured by the **WSDL2Java** command when the `webservices.xml` file is generated.
15. Locate the **Service implementation bean** field.
    a. Configure this field to indicate the EJB or servlet that implements the Web service. Configure by selecting **EJB link** for an enterprise bean module or **Servlet link** for a Web module. Use the drop down list in the **Service implementation bean** field to select the enterprise bean or servlet used to implement the Web service. The choices in the drop down menu come from the enterprise beans defined in the `ejb-jar.xml` file for an enterprise bean module or the servlets defined in the `web.xml` file for a Web module.

Configuring the ibm-webservices-bnd.xmi deployment descriptor

Develop implementation templates and bindings for the `ibm-webservices-bnd.xmi` from the Web Services Description Language (WSDL) file.

Do one of the following based on whether your implementation is an EJB Java archive (JAR) file or Web module Web archive (WAR) file:
- Assemble a Web services-enabled JAR file when starting from Java code.
- Assemble a Web services-enabled WAR file when starting from Java code.
- Assemble a Web services-enabled JAR file when starting from WSDL.
- Assemble a Web services-enabled WAR file when starting from WSDL.

This topic explains how to configure bindings using the Assembly Toolkit which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface. .

To configure the `ibm-webservices-bnd.xmi` deployment descriptor with the Assembly Toolkit:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file or WAR file into the Assembly Toolkit.

3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** > **J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the `webservices.xml` file in the **Project Navigator** pane.
6. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents appear.
7. Right-click the `webservices.xml` file.
8. Select **Open**. The **Web Services** editor opens.
9. Click the **Bindings** tab located at the bottom of the editor pane. The **Web Services Bindings** editor opens.
10. Verify the `wsdescNameLink` element settings.
    a. Expand the **Web services description bindings** section. Verify that the value of the `<webservice-description-name>` element in the `webservices.xml` deployment descriptor is listed in the section. If the value is not listed:
    b. (Optional) Click **Add**, choose the correct Web services name and click **OK**. You do not need to complete this step is you have verified that the correct Web services name is listed in the **Web Services Description Bindings** tab.
11. Verify the `pcnameLink` attribute settings.
    a. Expand the **Web Service Description Bindings** section. Verify that the correct service is selected. If the correct service is not listed:
    b. (Optional) Expand **Port Component Binding**. Verify the correct Web services name is selected in the **Web Service Description Bindings** section.

       This selection is a prerequisite to creating a `pcnameLink` attribute.
    c. In the **Port Component Binding** section, click **Add**. You need to make a selection in the **Web Service Description Bindings** section before you can create the port component binding in the **Port Component Binding** section. The **Port Component Bindings Dialog** opens.
    d. Select the desired port from the drop down list in the **PC Name Link** field.
    e. Click **OK**.
    f. Click the **Binding Configurations** tab to view the bindings for your port.
    g. **Save** the bindings file.
12. Click **File** > **Export** to export the JAR file, or continue using the Assembly Toolkit for configuration and assembly tasks.
13. Click **ctrl-s** to save your changes.

The `ibm-webservices-bnd.xmi` deployment descriptor is configured for the Web service implementation module.

ibm-webservices-bnd.xmi assembly properties

ibm-webservices-bnd.xmi properties

The `ibm-webservices-bnd.xmi` file is a deployment descriptor for a Web Services-enabled Web module or enterprise JavaBean (EJB) module. It contains information for the Web services runtime that is either WebSphere product-specific or was not specified by the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can edit these properties using the Assembly Toolkit. See Configuring the ibm-webservices-bnd.xmi deployment descriptor for instructions.

The following user-definable assembly properties are supported:

- **wsDescNameLink**

  Attribute of the wsdescBindings element that specifies the link to the corresponding <webservice-description-name> in webservices.xml.
- **pc-name-link**

  Attribute of the pcBindings element that specifies the link to the <port-component-name> in the webservices.xml file.
- **scope**

  Attribute of the pcBindings element that specifies when new instances of implementation beans are created. Possible values are Request, Session, and Application.

The value of scope for a deployed Web service can be changed using the administrative console. Using application management, navigate to the Web module of the Web service application and select Web Services Implementation Scope.

Example bindings file

The following examples demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID attribute whose value is a unique string. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings.

```
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
 xmlns:xmi="http://www.omg.org/XMI"
 xmlns:com.ibm.etools.webservice.wsbnd=
    "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
 <wsdescBindings wsDescNameLink="AddressBookService">
  <pcBindings pcNameLink="AddressBook" scope="Application"/>
 </wsdescBindings>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

Configuring the webservices.xml deployment descriptor for Handler classes

This topic explains how to use the Assembly Toolkit to configure the webservices.xml deployment descriptor for user-provided Handler classes. The Assembly Toolkit is a component of the Application Server Toolkit. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

You should have an enterprise archive (EAR) file for the applications you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes being configured. These classes implement the javax.xml.rpc.handler.Handler interface. For more information on writing handler classes, see Chapter 6 of the Web Services for J2EE 1.0 specification and chapter 12 of the JAX-RPC 1.0 specification available through Web services: Resources for learning. The application modules must contain the webservices.xml(for server) and webservicesclient.xml (for client) deployment descriptors.

To configure a handler in the `webservices.xml` deployment descriptor:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** and import the EAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Click the **Project Navigator** tab to switch to the **Project Navigator** pane.
5. Locate the project that contains the `webservices.xml` deployment descriptor. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents, including the `webservices.xml` file, are visible.
6. Right-click the `webservices.xml` file.
7. Click **Open**. The **Web Services** editor opens.
8. Expand the **Web services descriptions** section.
   a. Select the *service* for which you want to configure the handler.
9. Expand the **Port components** section.
10. Select a *port_component* for which you want the editor to open. The **Port Components** editor opens.
11. Expand the **Port component handlers** section.
12. Click **Add** at the bottom of the **Port component handlers** section. A **Class browser** opens.
13. Browse for the name of the Handler class in the module. When it displays in the **Matching types** field, select the class and click **OK**. The Class browser window closes after you click OK and the **Handlers** pane of the **Web Services Editor** opens.
14. (Optional) Configure properties in the **Handlers** pane. See Handler class properties for a list of the properties you can configure in this step.
15. Type **ctrl-s** to save the changes.

Developing a new Web service with an existing WSDL file using a Java bean

Locate the Web Services Description Language (WSDL) file that defines the Web service to be implemented. You can develop a WSDL or obtain one from an existing Web service through e-mail, downloading or a Uniform Resource Locator (URL).

To develop a new Web service with an existing WSDL file using a Java bean:

1. Develop Java bean implementation templates and bindings from a WSDL file.
2. Complete the Java bean implementation.
3. Assemble a Web services-enabled Web archive (WAR) file when starting from a WSDL file.
4. Configure the `webservices.xml` deployment descriptor.
5. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
6. Assemble a Web services-enabled WAR into an EAR file.
7. Deploy the EAR file into WebSphere Application Server.

Develop Web services deployment descriptor templates from a WSDL file.

You can either develop Web services deployment descriptor templates for a Java bean implementation or develop Web services deployment descriptor templates for an EJB implementation.

Developing Web services deployment descriptor templates for a Java bean implementation

To develop the Java bean implementation templates and bindings from a Web Services Description (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: file:*drive:\path\file_name*.wsdl. If you are using the UNIX platform, the URL looks like this: file:*/path/file_name*.wsdl. You can also specify local files using the absolute or relative file system path.

Implementation templates are generated using the -role develop-server option of the **WSDL2Java** command. The **WSDL2Java** command also generates bindings and deployment descriptors.

To develop Java bean implementation templates and bindings from a WSDL file: Run the **WSDL2Java -verbose -role develop-server -container web *wsdlURL*** command. Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

Templates for the implementation and deployment descriptors required to implement a Web service, as well as bindings files. These templates are partially filled with information from the WSDL file.

The following example uses an Java bean named AddressBook and a WSDL file named `AddressBook.wsdl`. After generating the template files from the **WSDL2Java -verbose -role develop-server -container web AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java..
WSWS3282I: Info: Generating WEB-INF\webservices.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating WEB-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating WEB-INF\ibm-webservices-ext.xmi.
```

The generated file named `AddressBookSOAPBindingImpl.java` is the template for the implementation bean. It is named after the port in the WSDL file. Generally, this class is renamed to a more meaningful name.

Complete the Java bean implementation.

Completing the Java bean implementation

Develop Java bean implementation templates and bindings from a Web Services Description Language (WSDL) file.

1. Edit the Java bean implementation template, *binding*Impl.java. *binding* is the name of the `<wsdl:binding>` element in the WSDL file.
   a. Complete the implementation of the methods in the template.
   b. (Optional) Make changes if necessary.
   c. (Optional) Change the class name if the binding name is not acceptable.

2. Compile all the Java classes.
3. Assemble a Web archive (WAR) file. Assemble all the Java classes into a WAR file using typical Web module assembly tools. Include all of the classes generated from running the **WSDL2Java** command tool when developing implementation templates and bindings from a WSDL file.

A Java archive (JAR) file containing a Java bean and supported classes created from the WSDL file.

Configure the `webservices.xml` deployment descriptor.

Developing a new Web service from an existing WSDL file using a stateless session enterprise bean

Set up a Web services development and unmanaged client execution environment.

Locate the Web Services Description Language (WSDL) file that defines the Web service to implement. The SOAP address URI is not required because it is updated when your new implementation is deployed.

Create the enterprise bean and artifacts that enable the enterprise bean to be a Web service and assemble those artifacts into the enterprise application as follows:
1. Develop implementation templates and bindings from a WSDL file.
2. Complete the enterprise bean implementation.
3. Assemble a Web services-enabled enterprise EJB Java archive (JAR) file.
4. Configure the `webservices.xml` deployment descriptor.
5. Configure the `ibm-webservices-bnd.xmi` deployment descriptor.
6. Assemble a Web services-enabled EJB JAR into an EAR file.
7. Enable the EAR file. When the EAR file contains EJB modules, it must have the Web services endpoint Web archive (WAR) file added with the **endptEnabler** command or Assembly Toolkit before deployment.
8. Deploy the EAR file into WebSphere Application Server.

An EJB implementation of a Web service defined in the WSDL file.

Developing EJB implementation templates and bindings from a WSDL file

To develop enterprise JavaBean (EJB) implementation templates and bindings from a Web Services Description (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: file:*drive:**path**file_name*.wsdl. If you are using the UNIX platform, the URL looks like this: file:*/path/file_name*.wsdl. You can also specify local files using the absolute or relative file system path.

Implementation templates are generated using the -role develop-server option of the **WSDL2Java** command.

Templates are generated for an EJB implementation for the following:
- EJB
- EJB remote interface

- EJB Home

The **WSDL2Java** command also generates bindings and deployment descriptors.

To develop implementation templates and bindings from a WSDL file:
Run the **WSDL2Java -verbose -role develop-server -container ejb** ***wsdlURL*** command. Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

Templates for the implementation and deployment descriptors required to implement a Web service, as well as bindings files. These templates are partially filled with information from the WSDL file.

The following example uses an enterprise bean named AddressBook and a WSDL file named `AddressBook.wsdl`. After generating the template files from the **WSDL2Java -verbose -role develop-server -container EJB AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookSoapBindingImpl.java.
WSWS3282I: Info: Generating addr\AddressBook_RI.java.
WSWS3282I: Info: Generating addr\AddressBookHome.java.
WSWS3282I: Info: Generating META-INF\webservices.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservices-ext.xmi.
```

Complete the EJB implementation.

Web services implementation scope

Use this page to view and manage the scope of the ports of a Web Service bean.

To view this administrative console page, click **Applications** >**Enterprise Applications** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Implementation Scope**.

Port

Specifies a port name for a Web service. A module can contain one or more Web services, each of which can contain one or more ports.

Web Service

Specifies the name of the Web service. A module can contain one or more Web services.

URI

Specifies the Uniform Resource Identifier (URI) of the binding file that defines the scope. The URI is relative to the Web module.

Scope

Specifies the scope of a port.
The scope determines when a new instance of a service implementation is created for the Web service ports in a module. An application scope causes the same instance of the implementation to be used for all requests on the application. A session scope causes the same instance to be used for all requests on each session. A request scope causes a new instance to be used on every request.

Default Port Mapping Definitions collection

Use this page to view and manage a default port type mapping for a Web service.

To view this page of the Administrative Console, click **Applications** >**Enterprise Application** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

For EJB modules, click **Applications** >**Enterprise Application** > *application_instance* > **EJB Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

Specify the default port of a service when a particular port type is requested. The port type is described by its local name and namespace. A getPort method specifying only the port type gets the port named by the default port local name and namespace.

Port Type Local Name

Specifies the name of this Web service.

Port Type Namespace

Specifies the local name describing the port type to be mapped.

Default Port Local Name

Specifies the namespace describing the port type to be mapped.

Default Port Namespace

Specifies the namespace of the port to map to.

Default Port Type Mapping Properties settings

Use this page to view and manage a default port type mapping for a Web service.

To view this page of the Administrative Console, click **Applications** >**Enterprise Application** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Client Bindings** > **Edit** > *default_port_instance*.

For EJB modules, click **Applications** >**Enterprise Application** > *application_instance* > **EJB Modules** > *module_instance*>**Web Services Client Bindings** > **Edit**> *default_port_instance*.

Specify the default port of a service when a particular port type is requested. The port type is described by its local name and namespace. A getPort method specifying only the port type gets the port named by the default port local name and namespace.

Port Type Local Name

Specifies the local name of the port type to be mapped.

Port Type Namespace

Specifies the namespace of port type to be mapped.

Default Port Local Name

Specifies the local name of the port to map to.

Default Port Namespace

Specifies the namespace of the port to map to.

Developing Web services clients based on Web Services for J2EE

This topic explains how to develop a Web services client based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

Before you begin this task, locate the Web Services Description Language (WSDL) file that defines the Web service to access.

To create the client code and artifacts that enable the application client to access a Web service:

1. Develop client bindings from a WSDL file. The client-side bindings and deployment descriptors are generated.
2. Complete the client implementation.
3. (Optional) Assemble a Web services-enabled client Java archive (JAR) file. Complete this step if you are developing a managed client that runs in the J2EE client container.
4. (Optional) Assemble a Web services-enabled client Web archive (WAR) file. Complete this step if you are developing a managed client that runs in the J2EE client container.
5. (Optional) Configure the `webservicesclient.xml` deployment descriptor. Complete this step if you are developing a managed client that runs in the J2EE client container.
6. (Optional) Configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor. Complete this step if you are deploying a managed client that runs in the J2EE client container and you want to override the default client settings. See ibm-webservicesclient-bnd.xmi assembly properties for more information about the `ibm-webservicesclient-bnd.xmi` deployment descriptor.
7. Test the Web services-enabled client application.

You have created and tested a Web services client application. For step-by-step information see Example: Developing Web services clients based on Web Services for J2EE.

Example: Developing Web services clients based on Web Services for J2EE

This example takes you through the steps to develop a Web service client. The development process is based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) and the Java API for XML-based remote procedure call (JAX-RPC) specification. For a Java or J2EE application to act as a client of a Web service, you must map the WSDL file to the Java code. The JAX-RPC specification defines the mapping between a WSDL file, Java code and XML Schema types.

Steps for this example task

1. Obtain the Web Services Description Language (WSDL) document for the Web service that you want to access.

   You can obtain the WSDL document from the service provider by e-mail or by looking it up in a Universal Description, Discovery and Integration (UDDI) registry.
2. Develop client bindings from your WSDL file.

   The WSDL document is used to generate all the information needed to invoke the Web service, including the Service Endpoint Interface and implementations;

generated service interface; `webservicesclient.xml` and `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` deployment descriptors.

The **WSDL2Java** command-line tool is run against your WSDL file to develop client bindings.

3. Implement the client.

   See Chapter 4 of the JSR-109 specification. You can access the specification through Web services: Resources for learning.

   You can also review the GetQuote sample available in the Samples Gallery.

4. Assemble the module.

   Assemble the client JAR file into an EAR file or assemble the client WAR file into an EAR file.

5. Configure the deployment descriptors.

   Configure the webservicesclient.xml deployment descriptor.

   Configure the ibm-webservicesclient-bnd.xmi deployment descriptor.

6.
7. Test the Web services client.

   You should test the client to make sure it correctly operates and binds to the Web service.

Developing client bindings from a WSDL file

To develop the client bindings from a Web Services Description (WSDL) file, you must obtain the Uniform Resource Locator (URL) of the WSDL file to use.

If it is a local file and you are running the Windows platform, the URL looks like this: file:*drive:\path\file_name*.wsdl. If you are using the UNIX platform, the URL looks like this: file:*/path/file_name*.wsdl. You can also specify local files using the absolute or relative file system path.

Client bindings are generated using the -role develop-client option in combination with the -container option of the WSDL2Java command. The -container option takes the following parameters:

- **-container client**

  Generates bindings and deployment descriptors for a client residing the application client container.
- **-container ejb**

  Generates bindings and deployment descriptors for a client that is an EJB in the EJB module.
- **-container web**

  Generates bindings and deployment descriptors for a client residing in the Web container.

To develop client bindings from a WSDL file:
Run the **WSDL2Java -verbose -role develop-client -container** ***type wsdlURL*** command.

Where *type* is **ejb** for an enterprise JavaBean (EJB) client, **web** for a Java bean client, or **client** for an application client.

Note: You can have:
- -container web

- -container ejb
- -container client

Since the **verbose** option is specified, a list of all generated files is displayed when the command runs.

The bindings and deployment descriptors needed by a client to use a Web service.

The following example uses an enterprise bean named AddressBook and a WSDL file named AddressBook.wsdl. After generating the bindings from the **WSDL2Java -verbose -role develop-client -container client AddressBook.wsdl** command, the following files are generated:

```
Parsing XML file:  file:e:/example/app/topdown/step1/AddressBook.wsdl
WSWS3185I: Info: Parsing XML file:  AddressBook.wsdl
WSWS3282I: Info: Generating addr\Address.java.
WSWS3282I: Info: Generating addr\Phone.java.
WSWS3282I: Info: Generating addr\StateType.java.
WSWS3282I: Info: Generating addr\AddressBook.java.
WSWS3282I: Info: Generating addr\AddressBookService.java.
WSWS3282I: Info: Generating META-INF\webservicesclient.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-bnd.xmi.
WSWS3282I: Info: Generating META-INF\AddressBook_mapping.xml.
WSWS3282I: Info: Generating META-INF\ibm-webservicesclient-ext.xmi.
```

Complete the client implementation.

Assemble a Web services-enabled client JAR and EAR file.

Assembling a Web services-enabled client JAR file into an EAR file

You need the following artifacts:
- Assembled client module, containing the implementation, all classes generated by the **WSDL2Java** command-line tool, MANIFEST.MF and deployment descriptor. This module can be:
  - An application client module containing META-INF/application-client.xml
  - An enterprise JavaBean (EJB) module containing META-INF/ejb-jar.xml
- Web Services Description Language (WSDL) file used to develop the client
- Templates for webservicesclient.xml and ibm-webservicesclient-ext.xmi deployment descriptors, if used.
- Generated JAX-RPC mapping deployment descriptor

You can use the Assembly Toolkit to assemble Web service-enabled client applications.

To assemble the client code and artifacts that enable the application client to access a Web service:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file, App Client JAR file, or WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project for the file you just imported in the **Project Navigator** pane.

6. Expand the `ejbModule` (for an EJB JAR file) or the `appClientModule` (for the application client JAR file) entry so the `META-INF` directory is displayed. Expand the `META-INF` directory.
7. Right-click the `META-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `META-INF` directory.
   a. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `META-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and the `ibm-webservicesclient-bnd.xmi` file in the `META-INF` subdirectory, if used.
8. Assemble the JAR file into an EAR file using typical assembly techniques if the client runs in a container.
9. Right-click on the `WEB-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `WEB-INF` directory.
   a. Copy the WSDL file to the `WEB-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `WEB-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file in the `WEB-INF` subdirectory, if used.

The artifacts required to enable the client module to use Web services are added to the module.

This example uses a JAR file named `AddressBookClient.jar` and an EAR file named `AddressBookClient.ear`:

```
META-INF/MANIFEST.MF
META-INF/application-client.xml
META-INF/wsdl/AddressBook.wsdl
META-INF/webservicesclient.xml
META-INF/AddressBook_mapping.xml
com/ibm/websphere/samples/webservices/addr/Address.class
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the `AddressBookClient.jar` file into the `AddressBookClient.ear` file, the `AddressBookClient.ear` file contains the following files:

```
META-INF/MANIFEST.MF
AddressBookClient.jar
META-INF/application.xml
```

Configure the webservicesclient.xml deployment descriptor .

Assembling a Web services-enabled client WAR file into an EAR file

You need the following artifacts:
- Assembled client Web archive (WAR) module, containing the implementation, all classes generated by the **WSDL2Java** command-line tool, `MANIFEST.MF` and deployment descriptor.
- Web Services Description Language (WSDL) file used to develop the client
- Templates for `webservicesclient.xml`, `ibm-webservicesclient-bnd.xmi` and `ibm-webservicesclient-ext.xmi` deployment descriptors, if used.
- Generated JAX-RPC mapping deployment descriptor

You can use the Assembly Toolkit to assemble Web service-enabled client applications.

To assemble the client code and artifacts that enable the application client to access a Web service:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project for the file you just imported in the **Project Navigator** pane.
6. Expand the `webContemt` entry so the `WEB-INF` directory is displayed. Expand the `WEB-INF` directory.
7. Right-click on the `WEB-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `WEB-INF` directory.
   a. Copy the WSDL file to the `WEB-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
   b. Copy the `webservicesclient.xml` and the JAX-RPC mapping file in the `WEB-INF` subdirectory in the same manner you copied the WSDL file. The JAX-RPC mapping file is indicated by the `<jaxrpc-mapping-file>` element in the `webservicesclient.xml` file.
   c. (Optional) Place the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file in the `WEB-INF` subdirectory, if used.
8. Assemble the WAR file into an EAR file using typical assembly techniques.

The artifacts required to enable the client module to use Web services are added to the module.

This example uses a WAR file named `AddressBookWeb.war` and an EAR file named `AddressBook.ear`:

```
WEB-INF/MANIFEST.MF
WEB-INF/web.xml
WEB-INF/wsdl/AddressBook.wsdl
WEB-INF/webservicesclient.xml
WEB-INF/AddressBook_mapping.xml
WEB-INF/ibm-webservicesclient-ext.xmi (optional)
WEB-INF/ibm-webservicesclient-bnd.xmi
com/ibm/websphere/samples/webservices/addr/Address.class
```

```
com/ibm/websphere/samples/webservices/addr/AddressBook.class
com/ibm/websphere/samples/webservices/addr/AddressBookClient.class
com/ibm/websphere/samples/webservices/addr/AddressBookService.class
...other generated classes...
```

After assembling the AddressBookWeb.war file into the AddressBook.ear file, the AddressBook.ear file contains the following files:

```
WEB-INF/MANIFEST.MF
AddressBookWeb.war
WEB-INF/application.xml
```

Configure the webservicesclient.xml deployment descriptor .

Configuring the ibm-webservicesclient-bnd.xmi deployment descriptor

This topic explains how to configure the `ibm-webservicesclient-bnd.xmi` deployment descriptor file using the Assembly Toolkit, which replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To configure the`ibm-webservicesclient-bnd.xmi` deployment descriptor file:

1. Start the Assembly Toolkit
2. Locate the `webservicesclient.xml` file in the module.
3. Double-click the `webservices.xml` file to open the **Web Services Client** editor.
4. Access the **Web Services Client Bindings** editor through the **Client Binding** tab at the bottom of the editor pane.
5. Verify the `componentNameLink` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Expand the **Component scoped references** section.
   c. Click **Add**.
   d. Select the component scoped references defined in the `webservicesclient.xml` file from the list.
6. Verify the `serviceRefLink` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Click the **Services References** tab.
   c. Click **Add**.
   d. Select the service references defined in the `webservicesclient.xml` file from the list.
7. Verify the `deploydWSDLFile` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Select the service references or component scoped reference desired.
   c. Expand the **Service reference details** section.
   d. Click **Browse** on the **Deployed WSDL file** field.
   e. Select the new WSDL file.
   f. Click **OK**.

   The `deployedWSDLFile` element of a deployed Web service can also be changed using the administrative console. Using application management, navigate to the Web module or EJB module of the Web service application and select **Web Services Client Bindings**.

8. Verify the `defaultMappings` element settings.
   a. Open the **Web Services Client Bindings** editor.
   b. Click **Default Mappings**.
   c. Click **Add**.
   d. Edit the entries in the newly added row to establish a mapping between a *portType* and *port* in the WSDL file. There can only be one entry for each *portType*.
   e. Select the new WSDL file.
   f. Click **OK**.

   The `defaultMappings` of a deployed Web service can also be changed using the administrative console. Using application management, navigate to the Web module or EJB module of the Web service application and select **Web Services Client Bindings**.
9. Access the **Web Services Client Port Bindings** editor through the **Port Bindings** tab at the bottom of the editor pane.
10. Verify the `syncTimeout` element settings.
    a. Create a **Port Qualified Name Bindings** for the port.
    b. Open the **Web Services Client Bindings** editor.
    c. Confirm that a service reference is selected in either the **Component scoped references** or **Service references** section.
    d. Expand the **Port qualified name bindings** section.
    e. Click **Add**. A new entry is now added to the **Port qualified name bindings** list.
    f. Click the new **Port qualified name bindings** entry. The **Web Services Client Port Bindings** editor opens.
    g. Expand the **Port qualified name bindings details** section.
    h. Type the *namespace* of the WSDL file port you want to configure, in the **Port Namespace Link** field.
    i. Type the *local_name* of the WSDL file port you want to configure in the **Port Local Name Link**field. The name displayed in the **Port qualified name bindings** list is the local name of the WSDL file port.
    j. Click **OK**.
    a. Configure the `syncTimeout` property by locating the **Synchronization timeout** field and enter the desired value.
11. Verify the `basicAuth` element settings.
    a. Open the **Web Services Client Bindings** editor.
    b. Expand the **Basic authentication** section.
    c. Type the desired value in the **User ID** and **Password** fields.
    d. Click **OK**.
12. Verify the `sslConfig` element settings.
    a. Open the **Web Services Client Bindings** editor.
    b. Expand the **SSL Configuration** section.
    c. Type the desired value in the **Name** field.
    d. Click **OK**.
13. After editing the properties, type **ctrl-s** on your keyboard to save the changes.

ibm-webservicesclient-bnd.xmi assembly properties

The `ibm-webservicesclient-bnd.xmi` file contains information for the Web services runtime that is WebSphere product-specific.

You can edit these properties using the Assembly Toolkit. See Configuring the ibm-webservicesclient-bnd.xmi deployment descriptor for instructions.

Assembly properties

The following user-definable assembly properties are supported:

- **`componentNameLink`**

  Attribute of the `componentScopedRefs` element that specifies the link to the corresponding `<component-scoped-refs>` element in `webservicesclient.xml` file. This property is used only when the Web service client is an EJB.

- **`serviceRefLink`**

  Attribute of the `serviceRefs` element that specifies the link to the `<service-ref-name>` in the `webservicesclient.xml` file.

  You can edit this property in the Assembly Toolkit:

- **`deployedWSDLFile`**

  Attribute of the `serviceRefs` element is optional and permits an alternate WSDL file to be used other than that specified in the `<wsdl-file>` element of `webservicesclient.xml` file. If this attribute is specified, the alternate WSDL file must be packaged in the same module and must be compatible with the development WSDL file. The deployedWSDLFile property is used to supply a new WSDL file containing a different endpoint URL than the original WSDL file.

- **`defaultMappings`** element

  Identifies which port should be used for a given portType when none is explicitly selected by the client. This element has the following attributes: `portTypeNamespace`, `portTypeLocalName`, `portNamespace`, `portLocalName`. These attributes identify which `wsdl:port` should be used for a `wsdl:portType`.

- **`syncTimeout`**

  Attribute of the `portQnameBindings` element that specifies how long, in seconds, to wait for a response from a synchronous call.

- **`basicAuth`**

  Element of the `portQnameBindings` element that can be used to authenticate a service client to the service endpoint, independent of the underlying transport that includes, HTTP, HTTPS, and JMS. Set the user ID and password attributes as needed.

- **`sslConfig`**

  Element of the `portQnameBindings` element that specifies the Secure Sockets Layer (SSL) configuration of an HTTPS outbound request. The name attribute is the name of a SSL configuration entry or alias defined in the SSL Configuration Repertoire.

  Note: This attribute is only used when the client is running in the WebSphere Application Server.

Example bindings file

The following example demonstrate the spelling and position of the various attributes. You cannot cut and paste these examples because they do not contain the required ID attributes. If you add elements to a binding file template generated by the **WSDL2Java** command, you must confirm that each element has an ID

attribute whose value is a unique string. Review the template xmi files generated by the **WSDL2Java** command for examples of ID strings.

```
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wscbnd=
   "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">

 <componentScopedRefs componentNameLink="myComponent ref"/>

 <serviceRefs serviceRefLink="myService ref"
    deployedWSDLFile="META-INF/wsdl/alternate.wsdl">
  <defaultMappings portTypeLocalName="AddressBook"
      portTypeNamespace="http://www.com.ibm" portLocalName="AddressBookPort"
      portNamespace="http://www.com.ibm"/>
   <portQnameBindings portQnameNamespaceLink="http://www.com.ibm"
         portQnameLocalNameLink="AddressBookPort" syncTimeout="99">
    <basicAuth userid="myId" password="myPassword"/>
    <sslConfig name="mynode/DefaultSSLSettings"/>
   </portQnameBindings>
  </serviceRefs>
 </com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Configuring the webservicesclient.xml deployment descriptor

You should have an enterprise JavaBean (EJB) Java archive (JAR) file, Web archive (WAR) file or an application client file that you can import into the Assembly Toolkit.

This topic explains how to configure the webservicesclient.xml deployment descriptor with the Assembly Toolkit. It is one of the tools available with the Application Server Toolkit product. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

To configure the webservicesclient.xml deployment descriptor with the Assembly Toolkit:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file, WAR file or application client file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** >**J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the webservicesclient.xml file in the **Project Navigator** pane.
6. Expand the directories under the project until the META-INF or WEB-INF directory and its contents appear.
7. Right-click on the webservicesclient.xml file.
8. Select **Open**. **The Web Services Client** editor opens.
9. Expand the **Service references** section.
10. Select the *service_reference* that you want to configure.
11. Expand the **Service reference overview** section.
12. Type the name of the service for which the client accesses in the **Description** field.
13. Expand the **Service reference implementation details** section.
    a. Type the name that the Java Naming Directory Interface (JNDI) uses to locate the service in the **Service references name** field. The JNDI lookup

string for this service is `java:comp/env/service-ref-name`. By convention, the service reference name always begins with `service/`.
   b. Type the class name, including package, of the generated Java interface that is the Service Interface for this Web service in the **Service interface name** field.
   c. Type the WSDL file name used by the client, relative to the root of the module, in the **WSDL file** field.
   d. Type the file name of the Java mapping file, relative to the root of the module, in the **JAX RPC mapping file** field.
14. Click **ctrl-s** to save the changes.

The `webservicesclient.xml` deployment descriptor is configured.

Configuring the webservicesclient.xml deployment descriptor for Handler classes

This topic explains how to use the Assembly Toolkit to configure the `webservicesclient.xml` deployment descriptor for user-provided Handler classes. The Assembly Toolkit is a component of the Application Server Toolkit. For more information about completing tasks with the Assembly Toolkit, click **Help** > **Help** in the Assembly Toolkit graphical user interface (GUI).

You should have an Enterprise archive (EAR) file for the applications you want to configure. For some handler use, such as logging or tracing, only the server or client application needs to be configured. For other handler use, including sending information in SOAP headers, the client and server applications must be configured with symmetrical handlers.

The modules in the EAR file should contain the handler classes being configured. These classes implement the `javax.xml.rpc.handler.Handler` interface. For more information on writing handler classes, see Chapter 6 of the Web Services for Java 2 platform, Enterprise Edition (J2EE) 1.0 specification and chapter 12 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification available through Web services: Resources for learning. The application modules must contain the `webservices.xml`(for server) and `webservicesclient.xml` (for client) deployment descriptors.

To configure a handler in the `webservicesclient.xml` deployment descriptor:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** and import the EAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** >**J2EE**.
4. Click the **Project Navigator** tab to switch to the **Project Navigator** pane.
5. Locate the project that contains the `webservicesclient.xml` deployment descriptor. Expand the directories under the project until the `META-INF` or `WEB-INF` directory and its contents, including the `webservicesclient.xml` file, are visible.
6. Right-click the `webservicesclient.xml` file.
7. Click **Open**. The **Service References** pane of the **Web Services Client** editor opens.
8. Expand the **Service references** section.
   a. Select the *service_reference* for which you want to configure the handler.

9. Expand the **Handlers** section.
10. Click **Add** at the bottom of the **Handlers** section. A **Class browser** opens.
11. Browse for the name of the Handler class in the module. When it displays in the **Matching types** field, select the class and click **OK**. The Class browser window closes after you click OK and the **Handlers** pane of the **Web Services Editor** opens.
12. (Optional) Configure properties in the **Handlers** pane. See Handler class properties for a list of the properties you can configure in this step.
13. Type **ctrl-s** to save the changes.

Handler class properties

You can configure the following Handler class properties through the Assembly Toolkit. See Configuring the webservices.xml deployment descriptor for Handler classes or Configuring the webservicesclient.xml deployment descriptors for Handler classes for instructions on how to configure the properties.

Description

Standard Java 2 platform, Enterprise Editioin (J2EE) technology descriptor field.

Display name

Standard J2EE technology descriptor field.

Small icon

Standard J2EE technology descriptor field.

Large icon

Standard J2EE technology descriptor field.

Handler name

The name of the handler. This name must be unique within the module.

Handler class

The fully qualified name of the Handler class. Initially, it is set by the Assembly Toolkit's class browser.

Initial parameters

Property names and values to be made available to the handler.

SOAP headers

Qnames of the SOAP headers that are processed by this handler. See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification, available through Web services: Resources for learning, for more information about setting this property.

SOAP roles

URIs containing the SOAP actor names for which the handler acts in the role of. See section 12.2.2 of the Java API for XML-based remote procedure call (JAX-RPC) 1.0 specification, available through Web services: Resources for learning, for more information about setting this property.

Example: Configuring Handler classes for Web services deployment descriptors

This scenario explains how to add trivial client and server Handler classes to a sample application named `WebServicesSamples.ear`. The Handler classes display messages when given a request or response to handle.

The code for the client Handler class is:

```
 package samples;

public class ClientHandler implements javax.xml.rpc.handler.Handler {
     public ClientHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ClientHandler: In handleFault");
   return true;  }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }

public void destroy() {
  }

public javax.xml.namespace.QName[] getHeaders() {
   return null; }
}
```

The code for the server Handler class is:

```
 package sample;
public class ServerHandler implements javax.xml.rpc.handler.Handler {
  public ServerHandler() { }

public boolean handleRequest(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleRequest");
   return true; }

public boolean handleResponse(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleResponse");
   return true; }

public boolean handleFault(javax.xml.rpc.handler.MessageContext
 context) {
   System.out.println("ServerHandler: In handleFault");
   return true; }

public void init(javax.xml.rpc.handler.HandlerInfo config) { }
```

```
public void destroy() {  }

public javax.xml.namespace.QName[] getHeaders() {
   return null;  }
}
```

1. Compile these classes using

   %JAVA_HOME%\bin\java -extdirs %WAS_EXT_DIRS% ClientHandler.java ServerHandler.java (on Windows)

   $JAVA_HOME/bin/java -extdirs $WAS_EXT_DIRS ClientHandler.java ServerHandler.java (on Unix)
2. Open the Assembly Toolkit and import the two sample EAR files:
   - %WAS_HOME%\samples\lib\WebServicesSamples\WebServicesSamples.ear on Windows or $WAS_HOME/samples/lib/WebServicesSamples/WebServicesSamples.ear on Unix.
   - %WAS_HOME%\samples\lib\WebServicesSamples\ApplicationClients.ear on Windows or $WAS_HOME/samples/lib/WebServicesSamples/ApplicationClients.ear on Unix..
3. Import the compiled handler classes into the projects for the sample modules:
   - Import sample.ClientHandler into the **appClientModule** directory of the **AddressBookClient** project.
   - Import sample.ServerHandler into the **ejbModule** directory of the **AddressBookW2JE** project.
4. Configure the webservicesclient.xml deployment descriptor for Handler classes.
5. Configure the webservices.xml deployment descriptor for Handler classes.
6. Save your changes and export the EAR files.
7. Uninstall the `WebServicesSamples.ear` application from your server if it is already installed.
8. Install the new `WebServicesSamples.ear` application.
9. Start the server.
10. Run the client:

    launchClient ApplicationClients.ear -CCjar=AddressBookClient.jar

    When the client executes, the console output is as shown below. The messages from the handlers are shown in bold.

    ```
    IBM WebSphere Application Server, Release 5.1
    J2EE Application Client Tool
    Copyright IBM Corp., 1997-2003
    WSCL0012I: Processing command line arguments.
    WSCL0013I: Initializing the J2EE Application Client
    Environment.
    WSCL0035I: Initialization of the J2EE Application Client
    Environment has completed.
    WSCL0014I: Invoking the Application Client class
    com.ibm.websphere.samples.webservices.addr.AddressBookClient
    >> Querying address for 'Purdue Boilermaker' using port
    AddressBookW2JE
    ClientHandler: In handleRequest
    ClientHandler: In handleResponse
    >> Response is:
           1 University Drive
           West Lafayette, IN 47907
    ```

```
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WE
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       2 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port
AddressBookJ2WB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       3 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
>> Querying address for 'Purdue Boilermaker' using port AddressBookW2JB
ClientHandler: In handleRequest
ClientHandler: In handleResponse
>> Response is:
       4 University Drive
       West Lafayette, IN 47907
       Phone: (765) 555-4900
```

For the client, the Handler class is configured for each service reference, not for each port. The `AddressBook` sample has four ports, but only one service reference, therefore the `ClientHandler` handles requests and responses on all ports.

When the server log file is examined, it contains:

```
[9/24/03 16:39:22:661 CDT] 4deec1c6 WebGroup      I SRVE0180I:
[HTTP router for AddressBookW2JE.jar] [/AddressBookW2JE] [Servlet.LOG]:
AddressBook: init
[9/24/03 16:39:23:161 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleRequest
[9/24/03 16:39:23:211 CDT] 4deec1c6 SystemOut     O ServerHandler: In handleResponse
```

What to do next

Install and test the application.

Testing Web services-enabled clients

Before testing your Java client, confirm that the server endpoint specified in the client Web Services Description Language (WSDL) file is running and available.

The following steps and examples assume that you are testing a system that has WebSphere Application Server installed, and that you have configured your environment as described in Setting up a Web services development and unmanaged client execution environment.

Tests are run differently depending on whether the client module has client container deployment information, which consists of the `application-client.xml` and `webservicesclient.xml` files, as well as the JAX-RPC mapping file and WSDL file. The client enterprise archive (EAR) files discussed in this topic are referred to as managed because they contain the deployment information. The client Java archive (JAR) files discussed are referred to as unmanaged because they that do not contain the deployment information.

To test Web services-enabled clients:

1. Test an unmanaged client JAR file.
   a. Execute your application with the **java** command. On Windows platforms:
      ```
      "%JAVA_HOME%\bin\java" "-Xbootclasspath/p:%WAS_BOOTCLASSPATH%"
      -Djava.security.auth.login.config="%WAS_HOME%\properties\wsjaas_client.conf"
      -Djava.ext.dirs="%WAS_EXT_DIRS%"
      -classpath "%WAS_CLASSPATH%;<list your application JAR files and classes>"
      <fully qualified class name to run><your application parameters>
      ```
      On UNIX:
      ```
      "$JAVA_HOME/bin/java" "-Xbootclasspath/p:$WAS_BOOTCLASSPATH"
      -Djava.security.auth.login.config="$WAS_HOME/properties/wsjaas_client.conf"
      -Djava.ext.dirs="$WAS_EXT_DIRS"
      -classpath "$WAS_CLASSPATH;<list of your application JAR files and classes>
      <fully qualified class name to run><your application parameters>
      ```
   The unmanaged client application runs.
2. Test a managed client EAR file.
   a. Execute your client application with the **launchClient** command. An example of using the command is as follows:
      ```
      launchClient clientEar
      ```

Web services-enabled clients that have been tested.

Troubleshoot your Web services application.

Web services client bindings

Use this page to specify the Web Service Description Language (WSDL) file name and default port type mappings for the Web services in a module.

To view this page, click **Applications** >**Enterprise Applications** > *application_instance* > **Web Modules** > *module_instance*>**Web Services Client Bindings**.

For EJB modules, click **Applications** >**Enterprise Applications** > *application_instance* > **EJB Modules** > *module_instance*>**Web Services Client Bindings**

Web Service

Specifies the name of this Web service. A module can contain one or more Web services.

URI

Specifies the Uniform Resource Identifier (URI) of the binding file that defines the scope. The URI is relative to the module.

WSDL Filename

Specifies the WSDL file name, which is relative to the module.

A Web service can specify the relative path within the module of a compatible WSDL file containing the actual URL to be used for requests. This is needed only if the original WSDL file does not contain a URL or when a different URL is needed. For a service endpoint with multiple ports defined, a default port mapping specifies the port to use for a port type.

Default Port Mappings

Specifies and manages the default port type mapping for a Web service when a particular port type is requested.

Assembling Web services applications based on Web Services for J2EE

This topic explains how to assemble a Web services application that is based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can assemble Web Services for J2EE modules with the Assembly Toolkit which replaces the Application Assembly Tool (AAT). The Assembly Toolkit is one of the tools available with the Application Server Toolkit product. .

To assemble Web services applications:
1. Start the Assembly Toolkit.
2. Assemble a Web services-enabled EJB JAR file.
3. Assemble a Web services-enabled EJB JAR file into an EAR file.
4. (Optional) Enable the EAR file. When the EAR file contains EJB modules, it must have the Web services endpoint WAR file added with the **endptEnabler** command-line tool or Assembly Toolkit before deployment.
5. Assemble a Web services-enabled WAR file.
6. Assemble a Web services-enabled WAR file into an EAR file.

A Web services-enabled EAR file that you can deploy into WebSphere Application Server.

Deploy the Web services-enabled EAR file into WebSphere Application Server.

Assembling a Web services-enabled EJB JAR file

You can assemble a Web services-enabled enterprise JavaBean (EJB) Java archive (JAR) file in one of two ways:
1. Assemble a Web services-enabled EJB JAR file when starting from Java code.
2. Assemble a Web services-enabled EJB JAR file when starting from Web Services Description Language (WSDL).

An assembled Web services-enabled EJB JAR file.

Configure the webservices.xml deployment descriptor .

Assembling a Web services-enabled EJB JAR file when starting from Java code

You need the following artifacts:
- Assembled Enterprise JavaBean (EJB) Java archive (JAR) file (not enabled for Web services)
- Compiled Java class for the Service Endpoint Interface
- Web Services Description Language (WSDL) file
- Complete `webservices.xml`,`ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi` and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

This topic explains how to assemble a Web service-enabled EJB JAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT) and is one of the tools available with the Application Server Toolkit product.

To assemble an Web services-enabled EJB JAR file when starting from Java code:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** > **J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project containing the JAR file you just imported in the **Project Navigator** pane.
6. Expand the `ejbModule` entry until the `META-INF` directory displays. Expand the `META-INF` directory.
7. Right-click the `META-INF` directory and click **New** > **Folder**. Create a subfolder named `wsdl` in the `META-INF` directory.
8. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
9. Copy the JAX-RPC mapping file, `webservices.xml`, `ibm-webservices-bnd.xmi`, and `ibm-webservices-ext.xmi` files into the `META-INF` directory.
10. Import the Service Endpoint Interface class so its package begins in the `ejbModule` directory. You can import either the source file or compiled class file. If you import the source file it automatically compiles.

The artifacts required to Web service-enable an EJB module for Web services are added to the JAR file.

After assembling a JAR file named `AddressBook.jar`, the JAR file contains the following files. The files added in this task are in bold:

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookBean.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

Configure the webservices.xml deployment descriptor .

Assembling Web services-enabled EJB JAR file when starting from WSDL

You need the following artifacts:

- An assembled Enterprise JavaBean (EJB) Java archive (JAR) file containing the EJB implementation and all classes generated by the **WSDL2Java** command tool when the **role** argument is develop-server and the **container** argument is EJB.
- A Web Services Description Language (WSDL) file
- Complete `webservices.xml`, `ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi`, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble Web services-enabled JAR files. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled EJB JAR file when starting from WSDL:

1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the EJB JAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** > **J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project for the JAR file you just imported in the **Project Navigator** pane.
6. Expand the `ejbModule` entry so the `META-INF` directory is displayed. Expand the `META-INF` directory.
7. Right-click the `META-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `META-INF` directory.
8. Copy the WSDL file to the `META-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
9. Copy the JAX-RPC mapping file as specified by the deployment descriptor `<jaxrpc-mapping-file>` element of `webservices.xml`.
10. Copy `webservices.xml`, `ibm-webservices-bnd.xmi` and `ibm-webservices-ext.xmi` into the `META-INF` subdirectory in the same manner.

The artifacts required to enable an EJB module for Web services are added to the JAR file.

After assembling a JAR file named `AddressBook.jar` contains the following files. The files added in this task are in bold:

```
META-INF/MANIFEST.MF
META-INF/ejb-jar.xml
addr/Address.class
addr/AddressBook_RI.class
addr/AddressBookSoapBindingImpl.class
addr/AddressBookHome.class
addr/Phone.class
addr/StateType.class
addr/AddressBook.class
META-INF/wsdl/AddressBook.wsdl
META-INF/ibm-webservices-bnd.xmi
META-INF/ibm-webservices-ext.xmi
META-INF/webservices.xml
META-INF/AddressBook_mapping.xml
```

Configure the webservices.xml deployment descriptor .

Assembling a Web services-enabled WAR file

You can assemble a Web services-enabled Web archive (WAR) file in one of two ways:

1. Assemble a Web services-enabled WAR file when starting from Java code.
2. Assemble a Web services-enabled WAR file when starting from WSDL.

A Web services-enabled WAR file is assembled.

Assembling a Web services-enabled WAR file when starting from Java code

You need the following artifacts:
- An assembled Web archive (WAR) file containing web.xml, but not Web services-enabled
- The Java class for the Service Endpoint Interface
- A Web Services Description Language (WSDL) file
- Complete webservices.xml,ibm-webservices-bnd.xmi, ibm-webservices-ext.xmi, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble a Web services-enabled WAR file. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file when starting from Java code:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** > **J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project for the WAR file you just imported in the **Project Navigator** pane.
6. Expand the WebContent directory so the WEB-INF directory is displayed. Expand the WEB-INF directory
7. Confirm that the WEB-INF/web.xml descriptor for the Web module contains a <servlet-class> element indicating the Java bean class that is implementing the service. Confirm by:
   a. Double-click **Web Deployment Descriptor**.
   b. In the **Web Deployment Descriptor** editor, click the **Servlets**.
   c. Enter the full path name of the Java bean class implementing the Web service in the **Servlet class** field.
   d. Close the editor window to save your changes.
8. Right-click the WEB-INF directory and click **New** > **Folder**. Create a subfolder named wsdl in the WEB-INF directory.
9. Copy the WSDL file to the WEB-INF\wsdl directory by right-clicking on the wsdl directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
10. Copy the JAX-RPC mapping file as specified by the deployment descriptor <jaxrpc-mapping-file> element of webservices.xml.
11. Copy webservices.xml,ibm-webservices-bnd.xmi and ibm-webservices-ext.xmi into the WEB-INF subdirectory in the same manner.
12. Import the Service Endpoint Interface class so that its package begins in the JavaSource directory. When you import the source file it is automatically compiled.

The artifacts required to Web service-enable the Web module are added to the WAR file.

Assemble a Web services-enabled WAR into an EAR file.

Assembling a Web services-enabled WAR file when starting from WSDL

You need the following artifacts:
- Assembled Web archive (WAR) file containing the enterprise JavaBean (EJB) implementation, all classes generated by the **WSDL2Java** command tool and a Web deployment descriptor, `web.xml`.
- A Web Services Description Language (WSDL) file
- Complete `webservices.xml`, `ibm-webservices-bnd.xmi`, `ibm-webservices-ext.xmi`, and Java API for XML-based remote procedure call (JAX-RPC) mapping deployment descriptors.

You can use the Assembly Toolkit to assemble Web services-enabled WAR files. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file when starting from WSDL:
1. Start the Assembly Toolkit.
2. Click **File** > **Import** to import the WAR file into the Assembly Toolkit.
3. Open the J2EE perspective by clicking **Windows** >**Open Perspective** > **Other** > **J2EE**.
4. Switch to the **Project Navigator** pane by clicking the **Project Navigator** tab.
5. Locate the project for the WAR file you just imported in the **Project Navigator** pane.
6. Expand the `WebContent` directory so the `WEB-INF` directory is displayed. Expand the `WEB-INF` directory
7. Confirm that the `WEB-INF/web.xml` deployment descriptor for the Web module contains a `<servlet>` element including the `<servlet-name>` element. To confirm:
   a. Double-click **Web Deployment Descriptor**.
   b. In the **Web Deployment Descriptor** editor click the **Servlets** tab.
   c. Enter the full path name of the Java bean class implementing the Web service in the **Servlet class** field.
   d. Close the editor window to save your change.
8. Right-click the `WEB-INF` directory and select **New** > **Folder**. Create a subfolder named `wsdl` in the `WEB-INF` directory.
9. Copy the WSDL file to the `WEB-INF\wsdl` directory by right-clicking on the `wsdl` directory and click **File** > **Import** > **File system**. Browse the WSDL file for this Web service and click **Finish**.
10. Copy the JAX-RPC mapping file as specified by the deployment descriptor `<jaxrpc-mapping-file>` element of `webservices.xml`.
11. Copy the `webservices.xml`, `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi` deployment descriptors in the `WEB-INF` subdirectory.

The artifacts required to Web service-enable the Web module is added to the WAR file.

Assemble a Web services-enabled WAR into an EAR file.

Assembling a Web services-enabled EJB JAR into an EAR file

Before assembling a Web services-enabled enterprise archive (EAR) file Assemble a Web services-enabled EJB Java archive (JAR) file.

You can assemble a Web services-enabled EAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled EAR file:

1. Start the Assembly Toolkit.
2. Assemble the Web services-enabled JAR file into an EAR file. The EAR file can contain an enterprise bean or application client JAR files, WAR files, Web applications, and metadata describing the applications or `application.xml` files.

A Web services-enabled EAR file.

In the following example, there is an `application.xml` deployment descriptor packaged with a Web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.jar
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
   "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
   "http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBookJ2WEE</display-name>
  <description>AddressBook EJB Example from Java</description>
  <module id="EjbModule_1">
   <ejb>AddressBook.jar</ejb>
  </module>
 </application>
```

Enable the EAR file. Then, deploy the EAR file into WebSphere Application Server.

Assembling a Web services-enabled WAR into an EAR file

Before assembling a Web services-enabled enterprise archive (EAR) fileAssemble a Web services-enabled Web archive (WAR) file.

This topic explains how to assemble a Web services-enabled WAR file into and EAR file using the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Tool (AAT) and is one of the tools available with the Application Server Toolkit product.

To assemble a Web services-enabled WAR file into an EAR file:

1. Start the Assembly Toolkit.
2. Assemble the Web services-enabled WAR file into an EAR file. Now assemble the EAR file that contains the JAR or WAR files. The EAR file can contain an enterprise bean or application client JAR files; Web applications or WAR files; and metadata describing the applications or `application.xml` files.

A Web services-enabled EAR file.

In the following example, there is an `application.xml` deployment descriptor packaged with a Web services-enabled JAR file called `AddressBook.jar` that is packaged into an EAR file called `AddressBook.ear`. The EAR file contains:

```
META-INF/MANIFEST.MF
META-INF/application.xml
AddressBook.war
```

An example of the `application.xml` deployment descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
 "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
 "http://java.sun.com/dtd/application_1_3.dtd">
 <application id="Application_ID">
  <display-name>AddressBook</display-name>
  <description>AddressBook Example from Java bean</description>
  <module id="WebModule_1">
   <web>
    <web-uri>AddressBook.war</web-uri>
    <context-root>/AddressBook</context-root>
   </web>
  </module>
 </application>
```

Deploy Web services based on Web Services for J2EE.

Enabling a Web services-enabled EAR file

Before doing this task, you need to Assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

You can add router modules to your Web services-enabled application, also known as an EAR file with the **endptEnabler** command or the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Toolkit (AAT) and is a component of the Application Server Toolkit (ASTK) product.

These tools add one or more router modules to the EAR file for each EJB JAR module within the EAR file. A router module provides an endpoint for the Web services in a particular enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

1. Enable an EAR file with the **endptEnabler** command-line tool.
2. Enable an EAR file with the Assembly Toolkit.

Deploy the EAR file into WebSphere Application Server.

Enabling a Web services-enabled EAR file with the endptEnabler command

Before doing this task, you need to assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

The **endptEnabler** command-line tool adds one or more router modules to the EAR file for each EJB JAR module within the EAR file. A router module provides an endpoint for the Web services in a particular Enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

To enable an EAR file with the **endptEnabler** command:

1. Invoke the **endptEnabler** command from the `install_root\bin` directory. If you are using UNIX, invoke the command from the `install_root/bin` directory.
2. Enter the name of the EAR file, when prompted.
3. Enter various input values as they are requested by the **endptEnabler** command. You are prompted for various input values for each Web services-enabled EJB JAR module in the EAR file. Typically, you should accept the defaults for each prompt. See endptEnabler prompts and commands for more information about **endptEnabler** command prompts.

An HTTP or JMS router module is added to the EAR file for each Web services-enabled EJB JAR module contained in the EAR file. For HTTP, a context-root is configured for the application so the Web service can be invoked through a URL. The URL used to invoke the Web service is:

```
http://host[:port]/context-root/services/port-component-name
```

Deploy the EAR file into WebSphere Application Server.

endptEnabler command: The IBM Web Services Developer Kit for z/OS contains the **endptEnabler** command-line tool needed for developing and implementing Web services. See Installing the IBM Web Services Developer Kit for z/OS to start using the tool.

The **endptEnabler** command enables a set of Web services within an enterprise archive (EAR) file. You can add one or more router modules to the EAR file that include a Web service-enabled EJB JAR file.

Each router module provides a Web service endpoint for a particular transport. For example, an HyperText Transport Protocol (HTTP) router module can be added so that the Web service can receive requests over the HTTP transport, and a Java Messaging Service (JMS) router module can be added so that the Web service can receive requests from a JMS queue or topic.

In its interactive mode, the **endptEnabler** command guides you through the required steps to enable one or more services within an application. The **endptEnabler** command makes a backup copy of your original EAR file in the event that you need to remove or add services at a later time. If your EAR file contains a Web service-enabled EJB JAR file, you must run the **endptEnabler** command before the EAR file is deployed. Otherwise, you do not need to run the command.

endptEnabler usage syntax

Invoke the **endptEnabler** command from the WebSphere Application Server `bin` directory. The command syntax is as follows:

```
endptEnabler
  [-verbose|-v]
  [-quiet|-q]
  [-help|-h|-?]
  [-properties|-p <properties-filename>]
  [-transport|-t <default-transports>]
  [-enableHttpRouterSecurity]
  [<ear-filename>]
```

All parameters are optional and described as follows:

- **-verbose, -v**

  Detailed progress messages are displayed as the tool processes the EAR file. This command-line option is mapped to the verbose global property.
- **-quiet, -q**

  No per-module progress messages are displayed as the tool processes the EAR file. This command-line option is mapped to the quiet global property.
- **-help, -h, -?**

  A brief help message is displayed explaining the various options.
- **-properties, -p <properties-filename>**

  Properties from the file <properties-filename> are read and used to control the behavior of the tool.
- **-transport, -t <default-transports>**

  Specifies the default list of transports for which router modules should be created for each EJB JAR file contained in the EAR file. This command-line option is mapped to the defaultTransports global property. Examples are:

  ```
  -transport http (the default)
  -transport jms
  -t http,jms
  ```
- **-enableHttpRouterSecurity**

  Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB's are secured in the EJB JAR file. This command-line option is mapped to the http.enableRouterSecurity global property.
- **<ear-filename>**

  Specifies the name of the EAR file to be processed.

  If the <ear-filename> parameter is not entered on the command line, the interactive mode is used. In interactive mode, you are prompted for the EAR file name, router module names and other important values as the processing occurs. The following dialog is an example of the endptEnabler interactive mode:

  Note: In this dialog, user input is in fixed width font, and endptEnabler output is in bold.

```
endptEnabler<enter>
WSWS2004I: IBM WebSphere Application Server Release 5
WSWS2005I: Web Services Enterprise Archive Endpoint Enabler Tool.
WSWS2007I: (C) COPYRIGHT International Business Machines Corp. 1997, 2003
WSWS2006I: Please enter the name of your EAR file: AddressBook.ear<enter>

WSWS2003I: Backing up EAR file to: AddressBook.ear~

WSWS2016I: Loading EAR file: AddressBook.ear
WSWS2017I: Found EJB Module: AddressBookEJB.jar

WSWS2029I: Enter http router name for EJB Module AddressBookEJB
 [AddressBookEJB_HTTPRouter.war]:<enter>
WSWS2030I: Enter http context root for EJB Module AddressBookEJB
 [/AddressBookEJB]:<enter>
```

```
WSWS2024I: Adding http router for EJB Module AddressBookEJB.jar.
WSWS2036I: Saving EAR file AddressBook.ear...
WSWS2037I: Finished saving the EAR file.
WSWS2018I: Finished processing EAR file AddressBook.ear.
```

If the <ear-filename> parameter is entered on the command line, the non-interactive mode is used. In non-interactive mode, router module names and other important values are determined from user-specified properties or default values.

endptEnabler properties

The **endptEnabler** command allows you to control its run time behavior by specifying a set of properties with the -properties command-line option. These properties fall into two categories: global and per-module. Global properties affect the overall behavior of the tool as it processes multiple EJB JAR modules within the EAR file. Per-module properties affect the processing of a particular EJB JAR module.

Global properties

The following table describes the global properties supported by the **endptEnabler** command:

Property name	Description	Default value
verbose	Displays detailed progress messages.	False
quiet	Displays only brief progress messages.	False
http.enableRouterSecurity	Enables you to add a security policy for all authenticated users to protect the HTTP router module if all the EJB's are secured in the EJB JAR file.	False
http.routerModuleNameSuffix	Specifies the suffix used to construct default HTTP router module names. The `.war` extension is added by the **endptEnabler** command.	_HTTPRouter
jms.routerModuleNameSuffix	Specifies the suffix used to construct default JMS router module names. The `.jar` extension is added by the **endptEnabler** command.	_JMSRouter
jms.listenerInputPortNameSuffix	Specifies the suffix used to construct default Listener Input Port names.	_ListenerPort
jms.defaultDestinationType	Specifies the default destination type to use for all JMS router modules added to the EAR file. This should be either queue or topic.	queue
defaultTransports	Specifies the default list of transports for which router modules should be created. The list can contain the values http and jms. Multiple values are separated by a comma. Examples are: http, jms and http,jms.	http

Per-module properties

The following table describes the per-module properties supported by the **endptEnabler** command. `<ejbJarName>` refers to the name of an EJB JAR module within the EAR file, without the `.jar` extension.

Property name	Description	Default value
<ejbJarName>.transports	Lists the transports for which router modules should be created for a particular EJB JAR file. The list can contain the values http and jms. Multiple values are separated by a comma. Examples are: http, jms and http,jms.	http
<ejbJarName>.http.skip	Specifies the flag which bypasses the addition of an HTTP router module even if it would otherwise be added (based on other properties). Valid values are true and false.	False
<ejbJarName>.http .routerModuleName	Specifies the name of the HTTP router module for a particular EJB JAR file.	<ejbJarName>_HTTPRouter
<ejbJarName>.http.contextRoot	Specifies the context root associated with the HTTP router module for a particular EJB JAR file.	/<ejbJarName>
<ejbJarName>.jms.skip	Specifies the Flag which bypasses the addition of an HTTP router module even if it would otherwise be added (based on other properties). Valid values are true and false.	false
<ejbJarName>.jms.routerModuleName	Specifies the name of the JMS router module for a particular EJB JAR file.	<ejbJarName>_JMSRouter
<ejbJarName>.jms .listenerInputPortName	Specifies the name of the Listener Input Port to be associated with the JMS router module.	<ejbJarName>_ListenerPort
<ejbJarName>.ejbJarName> .jms.destinationType	Specifies the JMS destination type associated with the JMS router. Valid values are queue and topic.	queue

Properties example

Suppose an EAR file contains an EJB JAR file named, `StockQuoteEJB.jar` that contains Web services. The following set of properties might be used to control the **endptEnabler** command runtime behavior as it processes the EAR file:

```
StockQuoteEJB.transports=http,jms

StockQuoteEJB.http.routerModuleName=StockQuoteEJB_HTTP

StockQuoteEJB.http.contextRoot=/StockQuote

StockQuoteEJB.jms.routerModuleName=StockQuoteEJB_JMS

StockQuoteEJB.jms.listenerInputPortName=StockQuote_LP

StockQuoteEJB.jms.destinationType=queue
```

endptEnabler examples

The following commands are examples of how the **endptEnabler** command can be used:

```
endptEnabler MyApp.ear

endptEnabler -t jms,http MyApp.ear

endptEnabler -v -properties MyApp.props MyApp.ear

endptEnabler -q -t jms MyApp.ear
```

Enabling a Web services-enabled EAR file with the Assembly Toolkit

Before doing this task, you need to Assemble a Web services-enabled EJB JAR into an enterprise archive (EAR) file.

You can add one or more router modules to your Web services-enabled application, also known as an EAR file with the Assembly Toolkit. The Assembly Toolkit replaces the Application Assembly Toolkit (AAT) and is a component of the Application Server Toolkit (ASTK) product.

A router module provides an endpoint for the Web services in a particular Enterprise JavaBean (EJB) Java archive (JAR) module.

Each router module supports a specific transport such as HyperText Transport Protocol (HTTP) or Java Messaging Service (JMS). If there are no EJB JAR modules in the EAR file, it is not necessary to use these tools.

To enable a Web services-enabled EAR file with the Assembly Toolkit:
1. Start the Assembly Toolkit.
2. Right-click on the EJB project to be enabled.
3. Click **Web Services** > **Endpoint Enabler**.
4. Specify the transport and router module names in the corresponding fields.
5. Click **OK**.

An HTTP or JMS router module is added to the EAR file for each Web services-enabled EJB JAR module contained in the EAR file. For HTTP, a context-root is configured for the application so the Web service can be invoked through a URL. The URL used to invoke the Web service is:

`http://`*`host[:port]`*`/`*`context-root`*`/services/`*`port-component-name`*

Deploy the EAR file into WebSphere Application Server.

Deploying Web services based on Web Services for J2EE

To deploy Web services that are based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification, you need an enterprise application, also known as an enterprise archive (EAR) file that has been configured and enabled for Web services. You can use either the administrative console or the **wsadmin** scripting interface to deploy an EAR file.

If you are installing an application containing Web services by using the **wsadmin** command, specify the **-deployws** option. If you are installing an application containing Web services by using the administrative console, select **Deploy**

WebServices during step 1 of the Install New Application wizard. For more information about installing applications using the administrative console see Installing a new application.

Note:

If the Web services in the application is previously deployed with the **wsdeploy** command, it is not necessary to specify Web services deployment during installation.

Use the following steps to deploy the EAR file with the **wsadmin** command:

1. Start `install_root\bin\wsadmin` from a command prompt. If you are using UNIX start `install_root/bin/wsadmin`.
2. Enter the **$AdminApp install** ***EARfile*** **"-usedefaultbindings -deployws"** command at the **wsadmin** prompt.

The Web service is installed into the application server.

Secure Web services.

wsdeploy command

This topic explains how to use the **wsdeploy** command-line tool with Web services that are based on the Web Services for J2EE specification. The **wsdeploy** command adds Websphere product-specific deployment classes to a Web services compatible enterprise application enterprise archive (EAR) file or an application client Java archive (JAR) file. These classes include:

- Stubs
- Serializers and deserializers
- Implementations of service interfaces

This deployment step must be performed at least once, and can be performed more than once. Deployment can be performed separately using the **wsdeploy** command, the Assembly Toolkit, or when the application is installed. The Assembly Toolkit replaces the Application Assembly Tool (AAT). It is one of the tools available with the Application Server Toolkit product. When using the **wsadmin** command for installation, specify the -deployws option. When using the administrative console for installation, select the **Deploy Web services** check box. When using the Assembly Toolkit, **Right-click** the module and select **Web Services** >**Deploy Web Services** from the pop-up menu. You can download the Assembly Toolkit from the Web site `http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en`.

The **wsdeploy** command operates as follows:

- Each module in the enterprise application or JAR file is examined
- If the module contains Web services implementations, indicated by the presence of the `webservices.xml` deployment descriptor, the associated Web Services Description Language (WSDL) files are located and the **WSDL2Java** command is run with the role deploy-server.
- If the module contains Web services clients, indicated by the presence of the `webservicesclient.xml` deployment descriptor, the associated WSDL files are located and the **WSDL2Java** command is run with the role deploy-client.
- The files generated by the **WSDL2Java** command are compiled and repackaged.

See WSDL2Java command for more information about the files that are generated for deployment.

When the generated files are compiled, they can reference application-specific classes outside the EAR or JAR file if the EAR or JAR file is not self-contained. In this case, use either the -jardir or -cp option to specify additional JAR or zip files to be added to CLASSPATH when the generated files are compiled.

wsdeploy command syntax

The command syntax is as follows:

```
wsdeploy Input_filename Output_filename [options]
```

Required options:

- ***Input_filename***

  Specifies the path to the EAR or JAR file to be deployed.
- ***Output_filename***

  Specifies the path of the deployed EAR or JAR file. If output_filename already exists, it is silently overwritten. The output_filename can be the same as the input_filename.

Other options:

- **-jardir** *directory*

  Specifies a directory containing JAR or zip files. All JAR and zip files in this directory are added to the CLASSPATH used to compile the generated files. This option can be specified zero or more times.
- **-cp** *entries*

  Specifies entries to be added to CLASSPATH when the generated classes are compiled. Multiple entries are separated the same as they would be in the CLASSPATH environment variable, with a semicolon on Windows platforms and a colon for UNIX platforms.
- **-codegen**

  Specifies that deployment code is to be generated, but not compiled. This option implicitly specifies the -keep option.
- **-debug**

  Includes debugging information when compiling, that is, use javac -g to compile.
- **-help**

  Displays a help message and exit.
- **-ignoreerrors**

  Do not stop deployment if validation or compilation errors are encountered.
- **-keep**

  Do not delete working directories containing generated classes. A message is displayed indicating the name of the working directory that is retained.
- **-novalidate**

  Do not validate the Web services deployment descriptors in the input file.
- **-trace**

  Displays processing information, including the names of the generated files.

Example

```
wsdeploy x.ear x_deployed.ear -trace -keep
Processing web service module x_client.jar.
Keeping directory: f:\temp\Base53383.tmp for module: x_client.jar.
Parsing XML file:f:\temp\Base53383.tmp\WarDeploy.wsdl
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java
Generating f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java
```

```
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeploy.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\WarDeployLocator.java.
Compiling f:\temp\Base53383.tmp\generatedSource\com\test\HelloWsBindingStub.java.
Done processing module x_client.jar.
```

Messages

- Flag *-f* is not valid

  Option *f* was not recognized as being a valid option.
- Flag *-c* is ambiguous

  Options may be abbreviated, but the abbreviation must be unique. In this case, the **wsdeploy** command can not determine which option was intended.
- Flag *-c* is missing parameter *-p*

  A required parameter for an option was omitted.
- Missing *p* parameter

  A required option was omitted.

Using the Java Messaging Service to transport Web services requests

WebSphere Application Server offers support for using a Java Messaging Service (JMS) transport layer, in addition to the existing HTTP transport. Using JMS transport allows your Web service clients and servers to communicate through JMS queues and topics instead of HTTP connections. One-way and synchronous two-way requests are supported.

Note: A Web service must be implemented as an enterprise JavaBean (EJB) to be accessed through the JMS transport.

The benefits of using JMS as an alternative to HTTP, include:
- Request and response messages are sent through reliable messaging.
- One-way requests allow clients and servers to be more loosely-coupled. For example, the server does not have to be active when the client sends the one-way request.
- One-way requests can be sent to multiple servers simultaneously through the use of a topic.

To use JMS as a transport for Web services requests:
1. Add a JMS binding and a SOAP address to the Web Services Description Language (WSDL) file. The WSDL file of a Web service must include a JMS binding and a SOAP address, which specifies a JMS endpoint URL string, in order to be accessible on the JMS transport. A JMS binding is a `wsdl:binding` element containing a `wsdlsoap:binding` element whose `transport` attribute ends in `soap/jms`, rather than the typical `soap/http` value.

   In addition to the JMS binding, a `wsdl:port` element referencing the JMS binding must be included in the `wsdl:service` element within the WSDL file. The `wsdl:port` element should contain a `wsdlsoap:address` element whose `location` attribute specifies a JMS endpoint URL string.

   Note: The specification of the actual JMS endpoint URL string can be deferred until you publish the WSDL file. When you develop the Web service, a placeholder such as `file:/unspecified_location` can be used for the endpoint URL string.
2. Decide on the names and types of JMS objects that your application uses Before your application can be installed, you need to:

a. Decide whether your Web service receives its requests from a queue or a topic.
b. Decide whether to use a secure destination (queue or topic) or a nonsecure destination.
c. Decide on the names for your destination, connection factory and listener port. The following list provides examples of the names that might be used for the mythical StockQuote Web service:
   - **Queue:** StockQuote_Q (JNDI name: jms/StockQuote_Q)
   - **Connection factory:** StockQuote_CF (JNDI name: jms/StockQuote_CF)
   - **Listener port:** StockQuoteEJB_ListenerPort
3. Define the JMS administered objects. Once you have decided on the names and types of the JMS objects, use the administrative console or the **wsadmin** scripting interface to define the JMS objects.
4. Add the JMS endpoints to your EAR file using the **endptEnabler** command tool. You must run the **endptEnabler** command to add a JMS endpoint or router module for each Web service-enabled EJB JAR file contained in the EAR file. By default, the **endptEnabler** command adds only HTTP endpoints, but the -transport jms option can be used to request the addition of JMS endpoints.
5. Deploy the Web services application. During the install process you are prompted for two types of information for each Web service-enabled EJB JAR contained in your EAR file:
   - The Java Naming and Directory Interface (JNDI) name of the connection factory to be used by the EJB JAR file message driven bean (MDB) listener for sending reply messages.

     If your Web service contains two-way operations, the MDB listener, defined inside the JMS endpoint added by **endptEnabler** command, needs to access a queue connection factory in order to add a reply message to the reply queue.

     The MDB listener uses a resource environment reference of `java:comp/env/jms/WebServicesReplyQCF`. Therefore, during the application install process, you must provide the actual JNDI name of the queue connection factory that should be used by the MDB listener for that Web service. You might want to use the same connection factory that you defined for use by clients in step 2.
   - The name of the listener port to be used by the MDB listener.

     A listener port is an object used to associate a JMS connection factory with a JMS destination (queue or topic). When deployed, an MDB is configured with the correct listener port so that messages from the desired queue or topic are properly delivered to the MDB. During deployment, you can modify the name of the listener port associated with each MDB listener. The listener port name contained in the input EAR file displays as a default value. If you specify the correct listener port name to the **endptEnabler** command, perhaps through the use of properties, during step 3, you can accept the default value. Otherwise, enter the correct listener port name.

     Hint: By default, the **endptEnabler** command produces listener port names of the form <ejb-jar-name>_ListenerPort. To simplify this step, define the listener ports that follow this naming convention during step 2.
6. Publish the WSDL file. In this step, you enter the JMS endpoint URL string to use for each Web service-enabled EJB JAR file belonging to the application. The JMS endpoint URLs are then written to the published WSDL files for use by clients.

   For example, suppose that an application called StockQuoteService contains an EJB JAR file named `StockQuoteEJB`, which contains one or more Web services accessible on the JMS transport. Suppose that, in step 2, you defined a queue

with the JNDI name `jms/StockQuote_Q` and a connection factory with the JNDI name `jms/StockQuote_CF` to be used by your application. In this example, you would specify the following string as the JMS URL prefix within the **Publish WSDL** user interface:

```
jms:/queue?destination=jms/StockQuote_Q&connectionFactory=jms/StockQuote_CF
```

The WSDL publisher uses this partial URL string to produce the actual JMS URL for each port component defined in the EJB JAR file. The published WSDL file can be used by clients needing to invoke the Web service.

Java Messaging Service endpoint URL syntax

A Java Messaging Service (JMS) endpoint URL is used to access a Web service with the JMS transport. This URL specifies the JMS destination and connection factory, as well as the port component name for the Web service request. This is similar to the HTTP endpoint URL, which specifies the host and port, as well as the context root and port component name.

A JMS endpoint URL has the following general form:

```
jms:/[queue|topic]?<property>=<value>&<property=<value>&...
```

The URL consists of the transport type, `jms:`, followed by either `/queue` or `/topic` to indicate the JMS destination type, followed by the query string containing a list of property and value pairs used to specify the JMS endpoint information.

The properties supported in the URL string are described as follows:

Destination-related properties (required)

Property name	Description
destination	Specifies the Java Naming and Directory Interface (JNDI) name of the destination queue or topic.
connectionFactory	Specifies the JNDI name of the connection factory.
targetService	Specifies the name of the port component to which the request is dispatched.

JNDI-related properties (optional)

Property name	Description
initialContextFactory	Specifies the name of the initial context factory to use which is mapped to the java.naming.factory.initial property.
jndiProviderURL	Specifies the JNDI provider URL which is mapped to the java.naming.provider.url property.

JMS-related properties (optional)

Property name	Description

deliveryMode	Indicates whether the request message should be persistent or not. The valid values are 1 for nonpersistent and 2 for persistent. The default value is 1.
timeToLive	Specifies the lifetime, in milliseconds, of the request message. A value of 0 indicates an infinite lifetime.
priority	Specifies the JMS priority associated with the request message. Valid values are between 0 to 9. The default value is 4.

The required properties, destination, connectionFactory, and targetService, must appear in the JMS endpoint URL string. The rest of the properties are optional.

You can set any of the properties on the client Stub object. This means that the various properties can be specified by including them as part of the endpoint URL or they can be set programmatically by the client on the Stub object. Properties specified on the client Stub object take precedence over properties specified as part of a JMS endpoint URL string.

Securing Web services based on WS-Security

Web services security for WebSphere Application Server is based on standards included in the Web services security (WS-Security) specification. These standards address how to provide protection for messages exchanged in a Web service environment. The specification defines the core facilities for protecting the integrity and confidentiality of a message and provides mechanisms for associating security-related claims with the message. Web services security is a message-level standard based on securing SOAP messages through XML digital signature, confidentiality through XML encryption, and credential propagation through security tokens.

Use the deprecated ″Securing Apache SOAP Web services″ topics in the WebSphere Application Server, Version 5 documentation if you are still using Apache SOAP Version 2.3.

To secure Web services, you must consider a broad set of security requirements, including authentication, authorization, privacy, trust, integrity, confidentiality, secure communications channels, federation, delegation, and auditing across a spectrum of application and business topologies. One of the key requirements for the security model in today’s business environment is the ability to interoperate between formerly incompatible security technologies (such as public key infrastructure, Kerberos and so on.) in heterogeneous environments (such as Microsoft .NET and Java 2 Platform, Enterprise Edition (J2EE)). The complete Web services security protocol stack and technology roadmap is described in Security in a Web Services World: A Proposed Architecture and Roadmap.

Specification: Web Services Security (WS-Security) proposes a standard set of SOAP extensions that you can use to build secure Web services. These standards confirm integrity and confidentiality, which are generally provided with digital signature and encryption technologies. In addition, Web services security provides a general purpose mechanism for associating security tokens with messages. A typical example of the security token is a user name and password token, in which a user

name and password are included as text. Web services security defines how to encode binary security tokens using methods such as X.509 certificates and Kerberos tickets.

An administrator can use any of the following methods to integrate message-level security into a WebSphere Application Server environment:

- Secure Web services using XML digital signature
- Secure Web services using XML encryption
- Secure Web services using basicauth authentication
- Secure Web services using identity assertion authentication
- Secure Web services using signature authentication
- Secure Web services using a pluggable token

Web services security specification- a chronology

This document describes the process used to develop the Web services security specifications.

Non-OASIS activities

In April 2002, IBM, Microsoft, and VeriSign proposed the *Web Services Security (WS-Security) specification* on their Web sites. This specification included the basic ideas of security token, XML signature, and XML encryption. The specification also defined the format for username tokens and encoded binary security tokens. After some discussion and an interoperability test based on the specification, the following issues were noted:

- The specification requires that the Web services security processors understand the schema correctly so that the processor distinguishes between the ID attribute for XML signature and XML encryption.
- The freshness of the message, which indicates whether the message complies with predefined time constraints, cannot be determined.
- Digested password strings do not strengthen security.

In August 2002, IBM, Microsoft, and VeriSign published the *Web Services Security Addendum*, which attempted to address the previously listed issues. The following solutions were put in the addendum:

- Require a global ID attribute for XML signature and XML encryption
- Use time stamp header elements that indicate the time of the creation, receipt, or expiration of the message
- Use password strings that are digested with a time stamp and nonce (randomly generated token)

OASIS activities

In June 2002, the Organization for the Advancement of Structured Information Standards (OASIS) received a proposed Web services security specification from IBM, Microsoft, and Verisign. The Web Services Security Technical Committee (WSS TC) was organized at OASIS soon after the submission. The technical committee included many companies including IBM, Microsoft, VeriSign, Sun Microsystems, and BEA Systems.

In September 2002, WSS TC published its first specification, *Web Services Security Core Specification, Working Draft 01*. This specification included the contents of both the original Web services security specification and its addendum.

The coverage of the technical committee became larger as the discussion proceeded. Since the Web Services Security Core Specification allows arbitrary types of security tokens, proposals were published as profiles. The profiles described the method for embedding tokens, including Security Assertion Markup Language (SAML) tokens and Kerberos tokens imbedded into the Web services security messages. Subsequently, the definitions of the usage for user name tokens and X.509 binary security tokens, which were defined in the original Web Services Security Specification, were divided into the profiles.

WebSphere Application Server supports the following specifications:

- Web Services Security: SOAP Message Security Draft 13 (formerly Web Services Security Core Specification)
- Web Services Security: Username Token Profile Draft 2

The following figure shows the various Web services security-related specifications. As indicated in the figure, the current support level for Web services security: SOAP message security is based on Draft 13 from May 2003. The current support level for Web services security User name token profiles, is based on Draft 2 from February 2003.

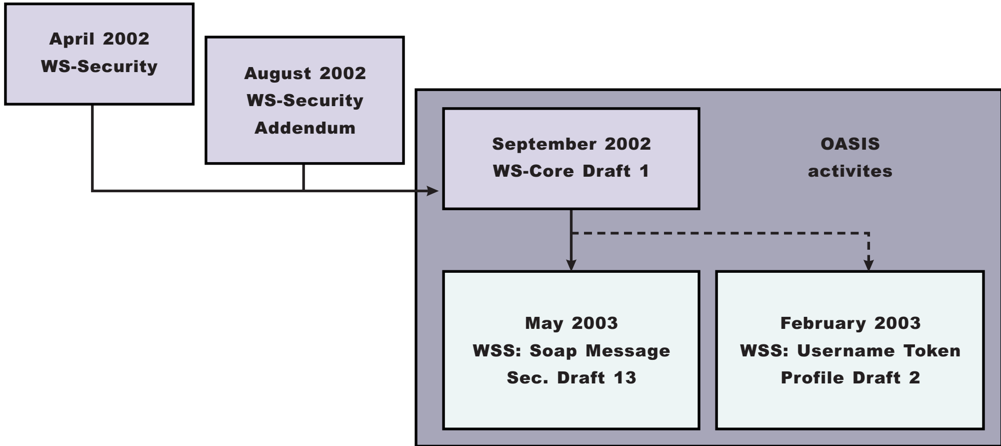


Figure 4. Web services security specification support

Web services security support

WebSphere Application Server, Versions 4.x, 5, and 5.0.1 support digital signature for Apache Simple Object Access Protocol (SOAP) Version 2.x. Beginning with WebSphere Application Server, Version 5.0.2, IBM supports *Web services security*, which is an extension of the IBM Web services engine to provide a quality of service. The IBM implementation is based on the Web services security specification, ″Web Services Security (WS-Security)″, originally proposed by IBM, Microsoft, and VeriSign in April 2002. Early versions of the proposed draft specification can be found in Web Services Security (WS-Security) Version 1.0 05 April 2002 and Web Services Security Addendum 18 August 2002. The WebSphere Application Server implementation is based on the Organization for the Advancement of Structured Information Standards (OASIS) working Draft 13 specification. (See the OASIS Web Services Security TC Web site for the latest working specification.) However, not all the features in the OASIS working Draft 13 specification are implemented.

WebSphere Application Server security infrastructure fully integrates Web services security with Java 2 Platform, Enterprise Edition (J2EE) security. When a user ID and password are embedded in a request message, authentication is performed with the user ID and password. If authentication is successful, a user identity is established and further resource access is authorized based on that identity. After the user ID and password are authenticated by the Web services security run time, a J2EE container performs authorization.

WebSphere Application Server provides an implementation of the key features of Web services security based on the following specifications:

- Specification: Web Services Security (WS-Security) Version 1.0 05 April 2002
- Web Services Security Addendum 18 August 2002
- Web Services Security: SOAP Message Security Working 13 May 2003
- Web Services Security: Username Token Profile Draft

The following table provides a summary of Web services security elements supported by WebSphere Application Server:

Table 1. Web services security elements

Element	Notes
`UsernameToken`	Both the user name and password for the `BasicAuth` authentication method and the user name for the identity assertion authentication method are supported. WebSphere Application Server, Version 5.0.2 does not support the Password Digest, Nonce, and Created attributes.
`BinarySecurityToken`	X.509 certificates and Lightweight Third Party Authentication (LTPA) can be embedded, but there is no implementation to embed Kerberos tickets. However, the binary token generation and validation are pluggable and are based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). You can extend this implementation to generate and validate other types of binary security tokens.
`Signature`	The X.509 certificate is embedded as a binary security token and can be referenced by the `SecurityTokenReference`. WebSphere Application Server does not support shared, key-based signature.
`Encryption`	Both the `EncryptedKey` and `ReferenceList` XML tags are supported. `KeyIdentifier` specifies public keys and `KeyName` identifies the secret keys. WebSphere Application Server has the capability to map an authenticated identity to a key for encryption or use the signer certificate to encrypt the response message.

Table 1. Web services security elements (continued)

Element	Notes
Timestamp	WebSphere Application Server supports the Created and Expires attributes. The freshness of the message, which indicates whether the message complies with predefined time constraints, is checked only if the Expires attribute is present in the message. WebSphere Application Server does not support the Received attribute, which is defined in the addendum. Instead, WebSphere Application Server uses the TimestampTrace Received attribute, which is defined in the OASIS specification.
XML based token	You can insert and validate an arbitrary format of XML tokens into a message. This format mechanism is based on the JAAS APIs.

Signing and encrypting attachments is not supported by WebSphere Application Server. The namespaces used for sending a message were published by OASIS in draft 13 (http://schemas.xmlsoap.org/ws/2003/06/secext). However, the Web services security run time in WebSphere Application Server can accept any of the following namespaces:

April 2002 specification
http://schemas.xmlsoap.org/ws/2002/04/secext

August 2002 addendum
http://schemas.xmlsoap.org/ws/2002/07/secext

http://schemas.xmlsoap.org/ws/2002/07/utility

OASIS draft published on draft 13 May 2003
http://schemas.xmlsoap.org/ws/2003/06/secext

http://schemas.xmlsoap.org/ws/2003/06/utility

WebSphere Application Server provides the following capabilities for Web services security:

- Integrity of the message
- Authenticity of the message
- Confidentiality of the message
- Privacy of the message
- Transport level security: provided by Secure Sockets Layer (SSL)
- Security token propagation (pluggable)
- Identity assertion

See the previous table titled, "Web services security elements," for a description of capabilities that are not supported.

Web services security and Java 2 Platform, Enterprise Edition security relationship

This document describes the relationship between Web services security (message level security) and Java 2 Platform, Enterprise Edition (J2EE) platform security.

WebSphere Application Server supports Java Specification Requests (JSR) 101 and JSR 109 (see Developing Web services for more information). These JSRs define Web services for the Java 2 Platform, Enterprise Edition (J2EE) architecture so that you can develop and run Web services on the J2EE component architecture. "Web services security" refers to the "Web services security: SOAP Message Security" specification (see Web services security support for more information).

Important

Note: "Web services security" refers to the "Web services security: SOAP Message Security" specification (see Web services security support for more information).

Securing Web services with WebSphere Application Server security (J2EE role-based security)

You can secure Web services using the existing security infrastructure of WebSphere Application Server, J2EE role-based security, and Secure Sockets Layer (SSL) transport level security.

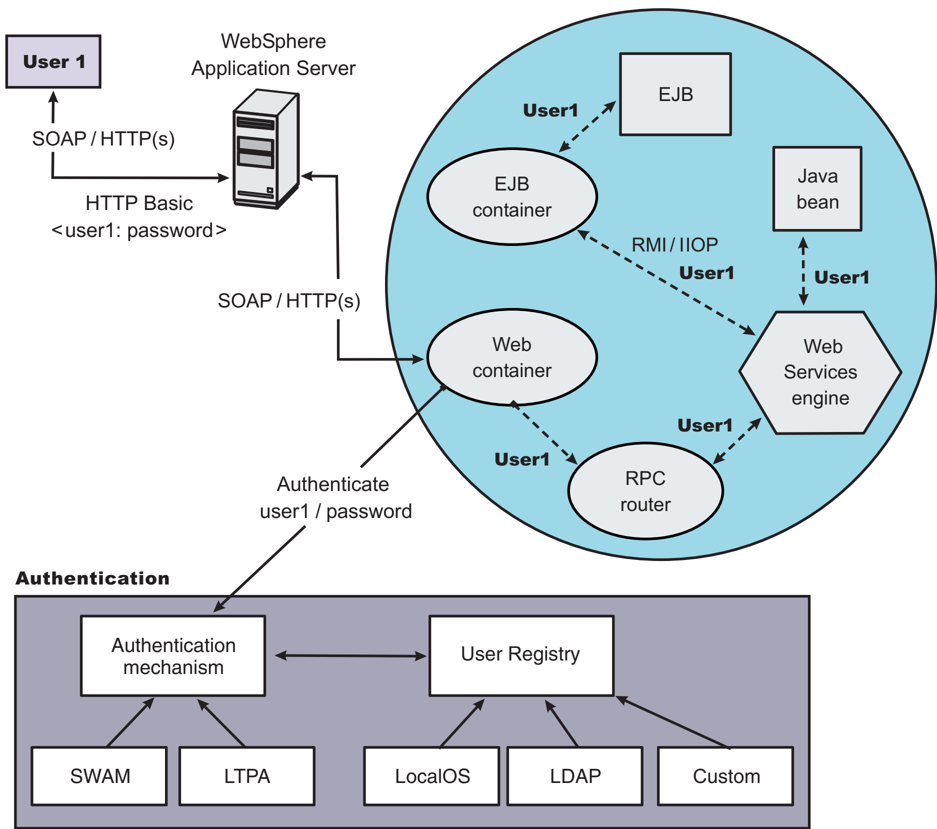


Figure 5. Simple object access protocol message flow using existing security infrastructure of WebSphere Application Server

The Web services endpoint can be secured using J2EE role-based security. The Web services sender sends the basic authentication data using the HTTP header. SSL (HTTPS) can be used to secure the transport. When the WebSphere Application Server receives the SOAP message, the Web container authenticates the user (in this example, `user1`) and sets the security context for the call. After the security

context is set, the SOAP router servlet sends the request to the implementation of the Web services (the implementation can be JavaBeans or enterprise bean files). For enterprise bean implementations, the EJB container performs an authorization check against the identity of `user1`.

The Web services endpoint also can be secured using the J2EE role. Then, authorization is performed by the Web container before the SOAP request is dispatched to the Web services implementation.

Securing Web services with Web services security at the message level

You can also secure Web services using Web services security at the message level. In this case, you can digitally sign or encrypt a certain part of the message. Web services security also supports security token propagation within the SOAP message. The following scenario assumes that the Web services endpoint is not secured with J2EE role-based security and the enterprise bean is secured with J2EE role-based security.

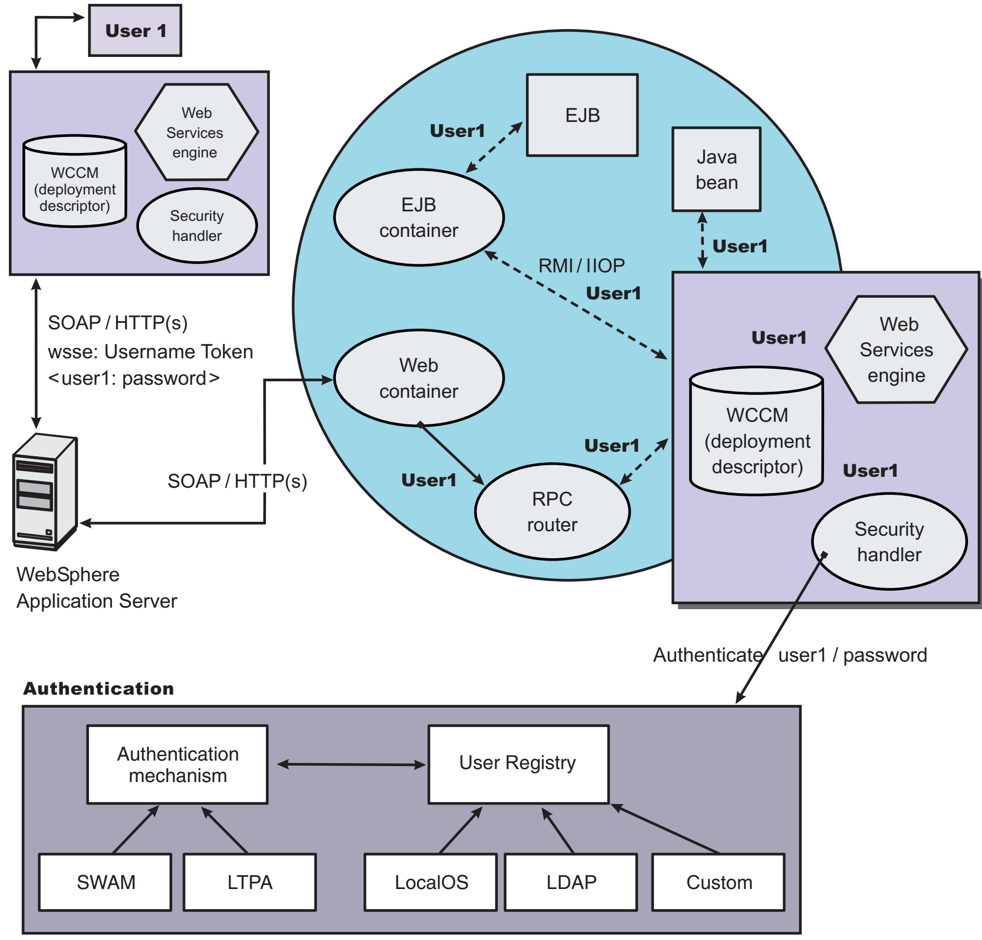


Figure 6. Simple Object Access Protocol message flow using Web services security

In this case, the Web services endpoint is not secured with J2EE role-based security. The Web services engine processes the SOAP message before the client sends the message to the Web services endpoint. The Web services security run time acts on

the security constraints, such as digitally signing, encrypting, or generating (and inserting) a security token in the SOAP header. In this case `<wsse:UsernameToken>` is generated using `user1` and the password. On the server-side (receiving), the Web services process the incoming message and Web services security enforces security constraints. This enforcement includes making sure that messages are properly signed, properly encrypted, and decrypted, authenticating the security token, and setting up the security context with the authenticated identity. (In this case, `user1` is the authenticated identity.) Finally, the SOAP message is dispatched to the Web services implementation (if the implementation is an enterprise beans file, the EJB container performs an authorization check against `user1`). SSL also might be used in this scenario.

Mixing the two

The second scenario shows that Web services security can complement J2EE role-based security. For example, SSL can be enabled at the transport level to provide a secure channel. Furthermore, if the Web services implementation is an enterprise beans file, you can leverage the EJB role-based authorization by performing authorization checks. Web services security run time leverages the security infrastructure to set the authenticated identity in the security context. The authenticated identity can be used in the downstream call to J2EE resources (or other resource types).

There are subtle consequences of combining the two scenarios. For example, if the HTTP transport is sending basic authentication data with `user1` and password in the HTTP header, but `<wsse:UsernameToken>` with `user99` and `letmein` also is inserted into the SOAP header. In the previous scenarios, there are two authentications performed. One authentication is performed by the Web container for authenticating `user1`, and the other is performed by Web services security for authenticating `user99`. The Web services security run time runs after the Web container runs and `user99` is the authenticated identity that is set in the security context.

Web services security can also propagate security tokens from the sender to the receiver for SOAP over a Java Message Service (JMS) transport.

Related concepts

"Web services security specification- a chronology" on page 384

"Web services security model in WebSphere Application Server"

Related tasks

"Web services security support" on page 385

"Developing Web services based on Web Services for J2EE" on page 293

Web services security model in WebSphere Application Server

The Web services security model used by WebSphere Application Server is the declarative model. WebSphere Application Server does not include any application programming interfaces (APIs) for programmatically interacting with Web services security. However, a few Server Provider Interfaces (SPIs) are available for extending some security-related behaviors.

Figure 7. Web services security model

The security constraints for Web services security are specified in IBM deployment descriptor extensions for Web services. The Web services security run time acts on the constraints to enforce Web services security for the Simple Object Access Protocol (SOAP) message. The scope of the IBM deployment descriptor extension is at the enterprise bean (EJB) or Web module level. Bindings are associated with each of the following IBM deployment descriptor extensions:

Client (Might be either a J2EE Client (Application Client Container) or Web services acting as a client)
`ibm-webservicesclient-ext.xmi`

`ibm-webservicesclient-bnd.xmi`

Server `ibm-webservices-ext.xmi`

`ibm-webservices-bnd.xmi`

It is recommended that you use the tools provided by IBM (the Assembly Toolkit and WebSphere Studio Application Developer) to create the IBM deployment descriptor extension and bindings. After the bindings are created, you can use the administrative console, the Assembly Toolkit, or the WebSphere Studio Application Developer to specify the bindings.

Important

Note: The binding information is collected after application deployment rather than during application deployment. The alternative is to specify the required binding information before deploying your application.

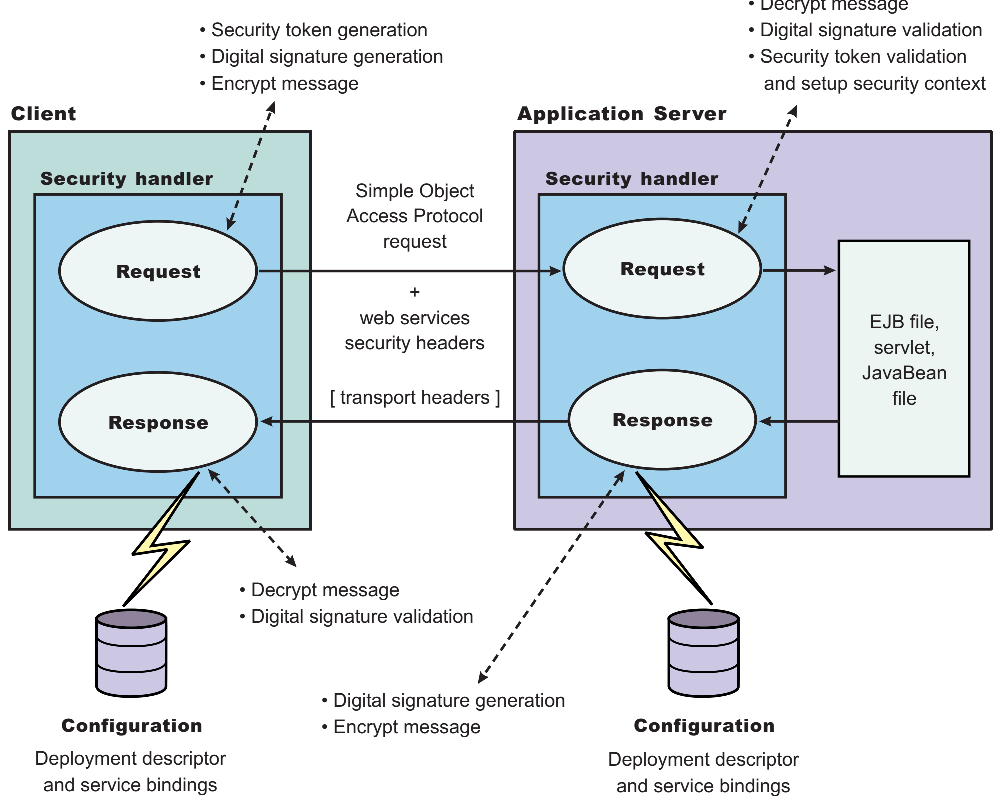


Figure 8. Web services security message interpretation

The Web services security run time enforces Web services security based on the defined security constraints in the deployment descriptor and binding files. Web services security has the following four points where it intercepts the message and acts on the security constraints defined:

Message points	Description
Request sender (defined in the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message (such as signing or encryption) before the message is sent, generating the time stamp or the required security token.
Request receiver (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Verifies that the Web services security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message if encrypted. • Validates the security tokens and sets up the security context for the downstream call.

Message points	Description
Response sender (defined in the `ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` files)	• Applies the appropriate security constraints to the SOAP message response, like signing the message, encrypting the message, or generating the time stamp.
Response receiver (defined in the `ibm-webservicesclient-ext.xmi` or `ibm-webservicesclient-bnd.xmi` files)	• Verifies that the Web services security constraints are met. • Verifies the freshness of the message based on the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints. • Verifies the required signature. • Verifies that the message is encrypted and decrypts the message, if encrypted.

Web services security property collection

Use this page to a view a list of additional properties for the configuration.

There are several ways to view a Web services security property collection panel. Complete the following steps to view one of these administrative console pages:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators >** *key_locator_name*.
3. Under Additional Properties, click **Properties**.
4. Click **New** to create a new property.
5. Click **Delete** to a delete a property that you specified previously.

Name

Specifies the name of the property.

Value

Specifies the value for the property.

Web services security property configuration settings

Use this page to configure additional properties.

There are several ways to view a Web services security property configuration settings panel. Complete the following steps to view one of these administrative console pages:

1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators >** *key_locator_name*.
3. Under Additional Properties, click **Properties > New**.

Property Name

Specifies the name of the property.

Data type:	String

Property Value

Specifies the value for the property.

Data type: String

Usage scenario for propagating security tokens

A sample scenario

This document describes a usage scenario for Web services security.

In scenario 1, Client 1 invokes Web services 1. Then Web services 1 calls EJB file 2. EJB file 2 calls Web services 3 and Web services 3 calls Web services 4.

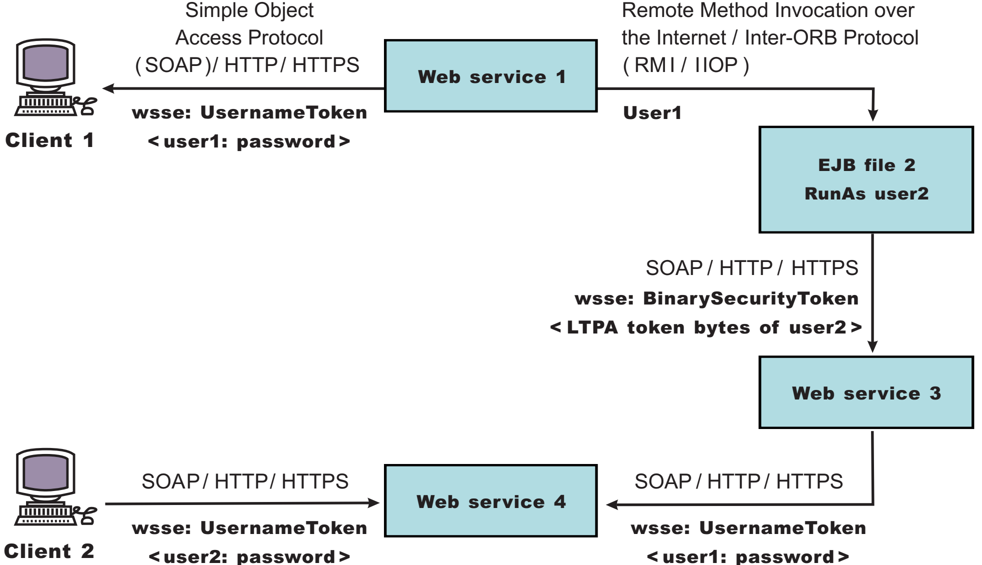


Figure 9. Propagating security tokens

The previous scenario shows how to propagate security tokens using Web services security, the security infrastructure of the WebSphere Application Server, and Java 2 Platform, Enterprise Edition (J2EE) security. Web services 1 is configured to accept `<wsse:UsernameToken>` only and use the BasicAuth authentication method. However, Web services 4 is configured to accept either `<wsse:UsernameToken>` using the BasicAuth authentication method or Lightweight Third Party Authentication (LTPA) as `<wsse:BinarySecurityToken>`. The following steps describe the scenario shown in the previous figure:

1. Client 1 sends a SOAP message to Web services 1 with `user1` and `password` in the `<wsse:UsernameToken>` element.
2. The `user1` and `password` values are authenticated by the Web services security run time and set in the current security context as the Java Authentication and Authorization Service (JAAS) Subject.
3. Web services 1 invokes EJB file 2 using the Remote Method Invocation over the Internet Inter-ORB Protocol (RMI/IIOP) protocol.
4. The `user1` identity is propagated to the downstream call.
5. The EJB container of EJB file 2 performs an authorization check against `user1`.
6. EJB file 2 calls Web services 3 and Web services 3 is configured to accept LTPA tokens.

7. The RunAs role of EJB file 2 is set to `user2`.
8. The LTPA CallbackHandler implementation extracts the LTPA token from the current JAAS Subject in the security context and Web services security run time inserts the token as `<wsse: BinarySecurityToken>` in the SOAP header.
9. The Web services security run time in Web services 3 calls the JAAS login configuration to validate the LTPA token and set it in the current security context as the JAAS Subject.
10. Web services 3 is configured to send LTPA security to Web services 4. In this case, assume that the RunAs role is not configured for Web services 3. The LTPA token of `user2` is propagated to Web services 4.
11. Client 2 uses the `<wsse:UsernameToken>` element to propagate the basic authentication data to Web services 4.

Web services security complements the WebSphere Application Server security run time and the J2EE role-based security. This scenario demonstrates how to propagate security tokens across multiple resources such as Web services and EJB files.

Configurations

The Web services security model used by WebSphere Application Server is the declarative model.

No Application Programming Interfaces (APIs) exist in WebSphere Application Server for programmatically interacting with Web services security. However, Service Provider Programming Interfaces (SPIs) are available for extending some security run-time behaviors. You can secure an application with Web services security by defining security constraints in the IBM extension deployment descriptors and in IBM extension bindings.

The development life cycle of a Web services security-enabled application is similar to the Java 2 Platform, Enterprise Edition (J2EE) model. See the following figure for more details.

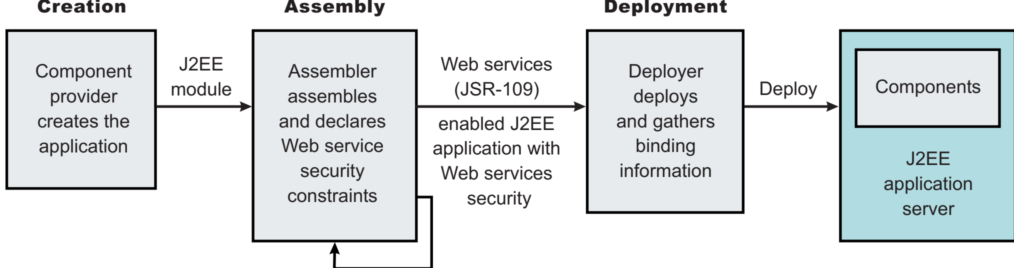


Figure 10. Application development life cycle

The Web services security constraints are defined by the assembler during the application assembly phase if the J2EE application is Web services-enabled. Create, define, and edit the Web services security constraints with the Assembly Toolkit, which can be downloaded from the following location: http://www.ibm.com /support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid =swg24005125&loc=en_US&cs=utf-8&lang=en+en

Web services security constraints

The security constraints for Web services security are specified in the IBM deployment descriptor extension for Web services. The assembler defines these constraints during the application assembly phase, if the J2EE application is Web services enabled. Define the Web services security constraints using the Assembly Toolkit.

The Web services security run time acts on the constraints to enforce Web services security for the SOAP message. The scope of the IBM deployment descriptor extension is at the EJB module or Web module level. There also are bindings associated with each of the following IBM deployment descriptor extensions:

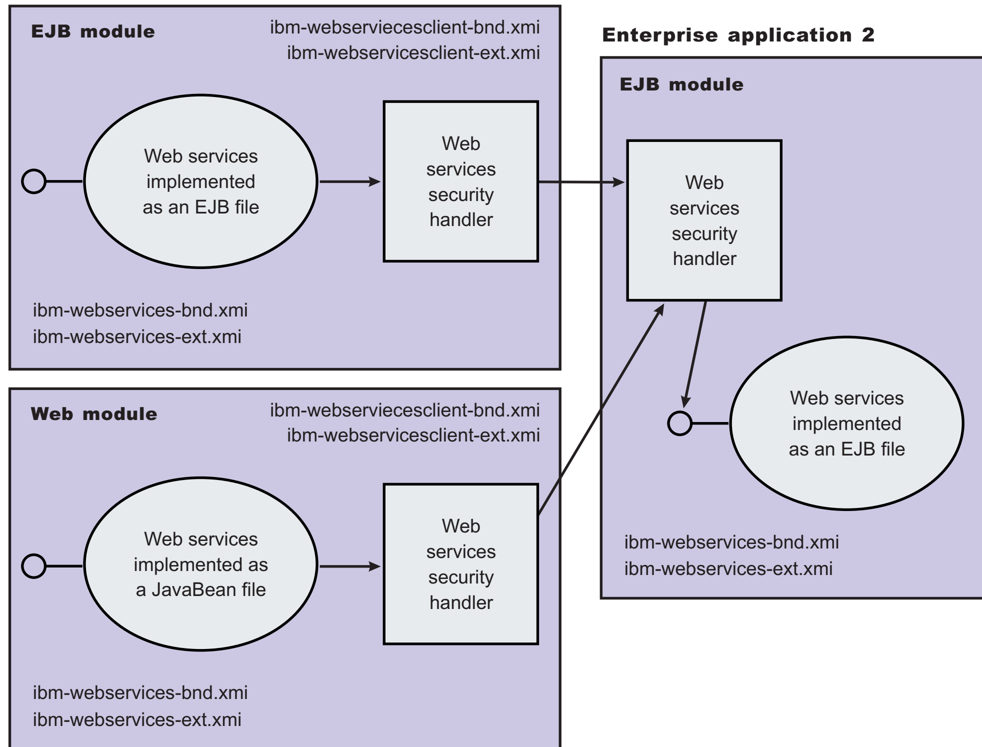
Client (might be either a J2EE client (application client container) or Web services acting as a client)

- `ibm-webservicesclient-ext.xmi`
- `ibm-webservicesclient-bnd.xmi`

Server

- `ibm-webservices-ext.xmi`
- `ibm-webservices-bnd.xmi`

The IBM extension deployment descriptor and bindings are associated with each EJB module or Web module. See Figure 2 for more information. If Web services is acting as a client, then it contains the client IBM extension deployment descriptors and bindings in the EJB module or Web module.



The Web services security handler acts on the security constraints defined in the IBM extension deployment descriptor and enforces the security constraints accordingly. There are outbound and inbound configurations in both the client and server security constraints.

In a SOAP request, the following message points exist:

- Sender outbound
- Receiver inbound
- Receiver outbound
- Sender inbound

These message points correspond to the following four security constraints:

- Request sender (sender outbound)
- Request receiver (receiver inbound)
- Response sender (receiver outbound)
- Response receiver (sender inbound)

The security constraints of request sender and request receiver must match. Also, the security constraints of the response sender and response receiver must match. For example, if you specify integrity as a constraint in the request receiver, then you must configure the request sender to have integrity applied to the SOAP message. Otherwise, the request is denied because the SOAP message does not include the integrity specified in the request constraint.

The four security constraints are shown in the following figure of Web services security constraints.

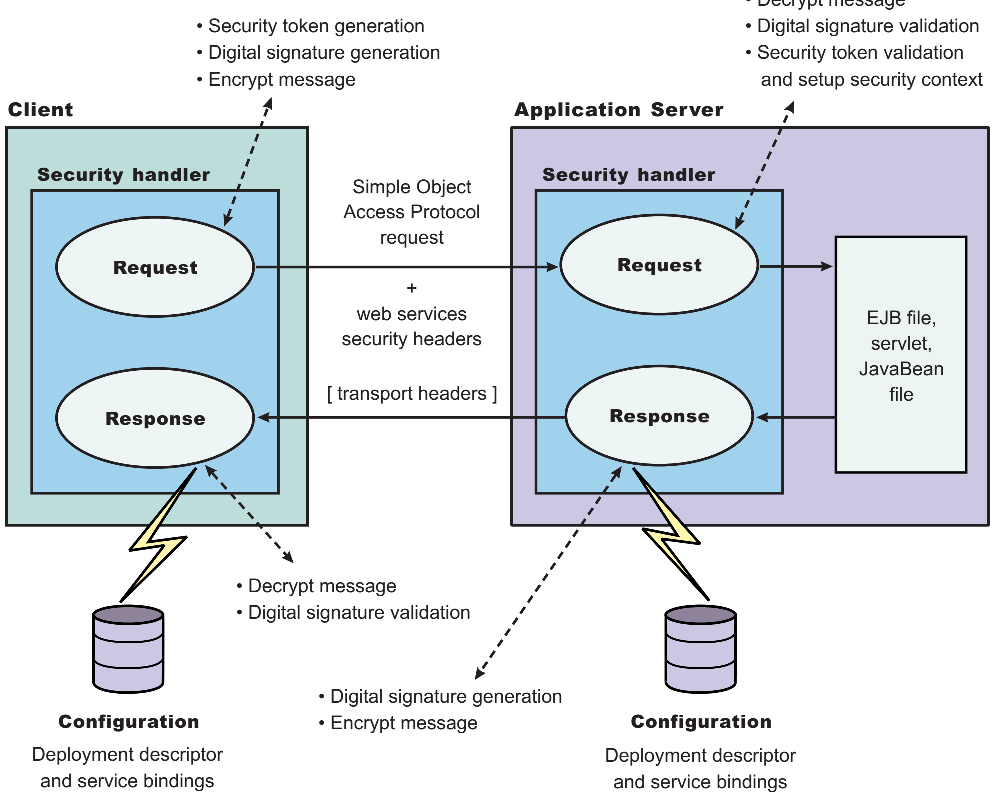


Figure 11. Web services security constraints

Sample configuration

WebSphere Application Server provides the following sample key stores for sample configurations. These sample key stores are for testing and sample purposes only. Do not use them a in production environment.

- `{USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks`
  - The keystore password is `client`
  - Trusted certificate with alias name, `soapca`
  - Personal certificate with alias name, `soaprequester` and key password `client` issued by intermediary certificate authority `Int CA2`, which is, in turn, issued by `soapca`
- `{USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-receiver.ks`
  - The keystore password is `server`
  - Trusted certificate with alias name, `soapca`
  - Personal certificate with alias name, `soapprovider` and key password `server`, issued by intermediary certificate authority `Int CA2`, which is, in turn, issued by `soapca`
- `{USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks`
  - The keystore password is `storepass`
  - Secret key `CN=Group1`, alias name `Group1`, and key password `keypass`
  - Public key `CN=Bob, O=IBM, C=US`, alias name `bob`, and key password `keypass`

- Private key CN=Alice, O=IBM, C=US, alias name alice, and key password keypass
- {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks
  - The keystore password is storepass
  - Secret key CN=Group1, alias name Group1, and key password keypass
  - Private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass
  - Public key CN=Alice, O=IBM, C=US, alias name alice, and key password keypass
- {USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer
  - The intermediary certificate authority is Int CA2.

Default binding (cell and server level)

WebSphere Application Server provides the following default binding information:

Trust anchors
Used to validate the trust of the signer certificate.
- SampleClientTrustAnchor is used by the response receiver to validate the signer certificate.
- SampleServerTrustAnchor is used by the request receiver to validate the signer certificate.

Collection Certificate Store
Used to validate the certificate path.
- SampleCollectionCertStore is used by the response receiver and the request receiver to validate the signer certificate path.

Key Locators
Used to locate the key for signature, encryption, and decryption.
- SampleClientSignerKey is used by the requesting sender to sign the SOAP message. The signing key name is clientsignerkey, which can be referenced in the signing information as the signing key name.
- SampleServerSignerKey is used by the responding sender to sign the SOAP message. The signing key name is serversignerkey, which can be referenced in the signing information as the signing key name.
- SampleSenderEncryptionKeyLocator is used by the sender to encrypt the SOAP message. It is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks keystore and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator.
- SampleReceiverEncryptionKeyLocator is used by the receiver to decrypt the encrypted SOAP message. The implementation is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator keystore key locator. The implementation is configured for symmetric encryption (DES or TRIPLEDES). However, to use it for asymmetric encryption (RSA), you must add the private key CN=Bob, O=IBM, C=US, alias name bob, and key password keypass.
- SampleResponseSenderEncryptionKeyLocator is used by the response sender to encrypt the SOAP response message. It is configured to use the {USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks keystore and the com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator key locator. This key locator maps an authenticated identity (of the current

thread) to a public key for encryption. By default, WebSphere Application Server is configured to map to public key `alice`, and you must change WebSphere Application Server to the appropriate user. The `SampleResponseSenderEncryptionKeyLocator` key locator also can set a default key for encryption. By default, this key locator is configured to use public key `alice`.

Trusted ID Evaluator

Used to establish trust before asserting to the identity in identity assertion.

`SampleTrustedIDEvaluator` is configured to use the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl` implementation. The default implementation of `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` contains a list of trusted identities. The list is defined as properties with `trustedId_*` as the key and the value as the trusted identity. Define this information for the server level in the administration console by completing the following steps:

1. Click **Servers > Application Servers >** *server1*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators >** *SampleTrustedIDEvaluator*

For the cell level, click **Security > Web Services > Trusted ID Evaluators >** *SampleTrustedIDEvaluator*.

Login Mapping

Used to authenticate the incoming security token in the Web services security SOAP header of a SOAP message.

- The BasicAuth authentication method is used to authenticate user name security token (user name and password).
- The signature authentication method is used to map a distinguished name (DN) into a WebSphere Application Server Java Authentication and Authorization Server (JAAS) Subject.
- The IDAssertion authentication method is used to map a trusted identity into a WebSphere Application Server JAAS Subject for identity assertion.
- The Lightweight Third Party Authentication (LTPA) authentication method is used to validate a LTPA security token.

The previous default bindings for trust anchors, collection certificate stores, and key locators are for testing or sample purpose only. Do not use them for production.

A sample configuration

The following examples demonstrate what IBM deployment descriptor extensions and bindings can do. The unnecessary information was removed from the examples to improve clarity. Do not copy and paste these examples into your application deployment descriptors or bindings. These examples serve as reference only and are not representative of the recommended configuration.

Use the following tools to create or edit IBM deployment descriptor extensions and bindings:

- Use the Assembly Toolkit to create or edit the IBM deployment descriptor extensions.

- Use the Assembly Toolkit or the administrative console to create or edit the bindings file.

The following example illustrates a scenario that:
- Signs the SOAP body, time stamp, and security token.
- Encrypts the body content and user name token.
- Sends the user name token (basic authentication data).
- Generates the time stamp for the request.

For the response, the SOAP body and time stamp are signed, the body content is encrypted, and the SOAP message freshness is checked using the time stamp. The freshness of the message indicates whether the message complies with predefined time constraints.

The request sender and the request receiver are a pair. Similarly, the response sender and the response receiver are a pair.

Tip: It is recommended that you use the WebSphere Application Server variables for specifying the path to the key stores. In the administrative console, click **Environment > Manage WebSphere Variables**. These variables often help with platform differences such as file system naming conventions. In the following examples, `${USER_INSTALL_ROOT}` is used for specifying the path to the key stores.

Client-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Sender
- Signs the SOAP body, time stamp and security token
- Encrypts the body content and user name token
- Sends the basic authentication token (user name and password)
- Generates the time stamp to expire in three minutes

Response Receiver
- Verifies that the SOAP body and time stamp are signed
- Verifies that the SOAP body content is encrypted
- Verifies that the time stamp is present (also check for message freshness)

Example 1: Sample client IBM deployment descriptor extension

The `xmi:id` statements are removed for readability. These statements must be added for this example to work.

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
   xmlns:com.ibm.etools.webservice.wscext=
    "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi">
  <serviceRefs serviceRefLink="service/myServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <clientServiceConfig actorURI="myActorURI">
        <securityRequestSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
```

```
          <references part="timestamp"/>
          <references part="securitytoken"/>
        </integrity>
        <confidentiality>
          <confidentialParts part="bodycontent"/>
          <confidentialParts part="usernametoken"/>
        </confidentiality>
        <loginConfig authMethod="BasicAuth"/>
        <addCreatedTimeStamp flag="true" expires="PT3M"/>
      </securityRequestSenderServiceConfig>
      <securityResponseReceiverServiceConfig>
        <requiredIntegrity>
          <references part="body"/>
          <references part="timestamp"/>
        </requiredIntegrity>
        <requiredConfidentiality>
          <confidentialParts part="bodycontent"/>
        </requiredConfidentiality>
        <addReceivedTimeStamp flag="true"/>
      </securityResponseReceiverServiceConfig>
    </clientServiceConfig>
  </portQnameBindings>
</serviceRefs>
</com.ibm.etools.webservice.wscext:WsClientExtension>
```

Client-side IBM extension bindings

Example 2 shows the client-side IBM extension binding for the security constraints described previously in the discussion on client-side IBM deployment descriptor extensions.

The signer key and encryption (decryption) key for the message can be obtained from the keystore key locator implementation (`com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`). The signer key is used for encrypting the response. The sample is configured to use the Java Certification Path API to validate the certificate path of the signer of the digital signature. The user name token (basic authentication) data is collected from the standard in (stdin) prompts using one of the default Java Authentication and Authorization Service (JAAS) implementations :`javax.security.auth.callback.CallbackHandler` implementation (`com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`).

Example 2: Sample client IBM extension binding

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscbnd:ClientBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wscbnd=
   "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscbnd.xmi">
  <serviceRefs serviceRefLink="service/MyServ">
    <portQnameBindings portQnameLocalNameLink="Port1">
      <securityRequestSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm=
            "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="clientsignerkey"
            locatorRef="SampleClientSignerKey"/>
          <canonicalizationMethod algorithm=
           "http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <keyLocators name="SampleClientSignerKey"
            classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}PDM2OjEr"
```

```
        path="${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
        type="JKS"/>
      <keys alias="soaprequester" keypass="{xor}PDM2OjEr"
        name="clientsignerkey"/>
    </keyLocators>
    <encryptionInfo name="EncInfo1">
      <encryptionKey name="CN=Bob, O=IBM, C=US"
         locatorRef="SampleSenderEncryptionKeyLocator"/>
      <encryptionMethod algorithm=
        "http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <keyEncryptionMethod algorithm=
        "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    </encryptionInfo>
    <keyLocators name="SampleSenderEncryptionKeyLocator"
        classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
      <keyStore storepass="{xor}LCswLTovPiws"
        path="${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-sender.jceks"
        type="JCEKS"/>
      <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
    </keyLocators>
    <loginBinding authMethod="BasicAuth"
      callbackHandler="com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
  </securityRequestSenderBindingConfig>
  <securityResponseReceiverBindingConfig>
    <signingInfos>
      <signatureMethod algorithm=
        "http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
      <certPathSettings>
        <trustAnchorRef ref="SampleClientTrustAnchor"/>
        <certStoreRef ref="SampleCollectionCertStore"/>
      </certPathSettings>
      <canonicalizationMethod algorithm=
        "http://www.w3.org/2001/10/xml-exc-c14n#"/>
      <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    </signingInfos>
    <trustAnchors name="SampleClientTrustAnchor">
      <keyStore storepass="{xor}PDM2OjEr"
        path="${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
        type="JKS"/>
    </trustAnchors>
    <certStoreList>
      <collectionCertStores provider="IBMCertPath"
          name="SampleCollectionCertStore">
        <x509Certificates path="${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer"/>
      </collectionCertStores>
    </certStoreList>
    <encryptionInfos name="EncInfo2">
      <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
      <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
      <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
    </encryptionInfos>
    <keyLocators name="SampleReceiverEncryptionKeyLocator"
        classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
      <keyStore storepass="{xor}PDM2OjEr"
        path="${USER_INSTALL_ROOT}/etc/ws-security/samples/dsig-sender.ks"
        type="JKS"/>
      <keys alias="soaprequester" keypass="{xor}PDM2OjEr" name="clientsignerkey"/>
    </keyLocators>
  </securityResponseReceiverBindingConfig>
 </portQnameBindings>
</serviceRefs>
</com.ibm.etools.webservice.wscbnd:ClientBinding>
```

Server-side IBM deployment descriptor extension

The client-side IBM deployment descriptor extension describes the following constraints:

Request Receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Verifies that the SOAP body, time stamp, and security token are signed.
- Verifies that the SOAP body content and user name token are encrypted.
- Verifies that the basic authentication token (user name and password) is in the Web services security SOAP header.
- Verifies that the time stamp is present (also check for message freshness). The freshness of the message indicates whether the message complies with predefined time constraints.

Response Sender (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi`)

- Signs the SOAP body and time stamp
- Encrypts the SOAP body content
- Generates the time stamp to expire in 3 minutes

Example 3: Sample server IBM deployment descriptor extension

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsext=
   "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi">
  <wsDescExt wsDescNameLink="MyServ">
    <pcBinding pcNameLink="Port1">
      <serverServiceConfig actorURI="myActorURI">
        <securityRequestReceiverServiceConfig>
          <requiredIntegrity>
            <references part="body"/>
            <references part="timestamp"/>
            <references part="securitytoken"/>
          </requiredIntegrity>
          <requiredConfidentiality">
            <confidentialParts part="bodycontent"/>
            <confidentialParts part="usernametoken"/>
          </requiredConfidentiality>
          <loginConfig>
            <authMethods text="BasicAuth"/>
          </loginConfig>
          <addReceivedTimestamp flag="true"/>
        </securityRequestReceiverServiceConfig>
        <securityResponseSenderServiceConfig actor="myActorURI">
          <integrity>
            <references part="body"/>
            <references part="timestamp"/>
          </integrity>
          <confidentiality>
            <confidentialParts part="bodycontent"/>
          </confidentiality>
          <addCreatedTimestamp flag="true" expires="PT3M"/>
        </securityResponseSenderServiceConfig>
      </serverServiceConfig>
    </pcBinding>
  </wsDescExt>
</com.ibm.etools.webservice.wsext:WsExtension>
```

Server-side IBM extension bindings

The following binding information reuses some of the default binding information defined either at the server level or the cell level, which depends upon the

installation. For example, request receiver is referencing the SampleCollectionCertStore certification store and the SampleServerTrustAnchor trust store is defined in the default binding. However, the encryption information in the request receiver is referencing a SampleReceiverEncryptionKeyLocator key locator defined in the application-level binding (the same ibm-webservices-bnd.xmi file). The response sender is configured to use the signer key of the digital signature of the request to encrypt the response using one of the default key locator (com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator) implementations.

Example 4: Sample server IBM extension binding

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsbnd:WSBinding xmi:version="2.0"
  xmlns:xmi="http://www.omg.org/XMI"
  xmlns:com.ibm.etools.webservice.wsbnd="
   http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsbnd.xmi">
  <wsdescBindings wsDescNameLink="MyServ">
    <pcBindings pcNameLink="Port1" scope="Session">
      <securityRequestReceiverBindingConfig>
        <signingInfos>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <certPathSettings>
            <trustAnchorRef ref="SampleServerTrustAnchor"/>
            <certStoreRef ref="SampleCollectionCertStore"/>
          </certPathSettings>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfos>
        <encryptionInfos name="EncInfo1">
          <encryptionKey locatorRef="SampleReceiverEncryptionKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfos>
        <keyLocators name="SampleReceiverEncryptionKeyLocator"
          classname="com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator">
          <keyStore storepass="{xor}LCswLTovPiws"
           path="${USER_INSTALL_ROOT}/etc/ws-security/samples/enc-receiver.jceks" type="JCEKS"/>
          <keys alias="Group1" keypass="{xor}NDomLz4sLA==" name="CN=Group1"/>
          <keys alias="bob" keypass="{xor}NDomLz4sLA==" name="CN=Bob, O=IBM, C=US"/>
        </keyLocators>
      </securityRequestReceiverBindingConfig>
      <securityResponseSenderBindingConfig>
        <signingInfo>
          <signatureMethod algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
          <signingKey name="serversignerkey" locatorRef="SampleServerSignerKey"/>
          <canonicalizationMethod algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          <digestMethod algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        </signingInfo>
        <encryptionInfo  name="EncInfo2">
          <encryptionKey locatorRef="SignerKeyLocator"/>
          <encryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
          <keyEncryptionMethod algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
        </encryptionInfo>
        <keyLocators name="SignerKeyLocator"
          classname="com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator"/>
      </securityResponseSenderBindingConfig>
    </pcBindings>
  </wsdescBindings>
  <routerModules transport="http" name="StockQuote.war"/>
</com.ibm.etools.webservice.wsbnd:WSBinding>
```

View Web services client deployment descriptor

Use this page to view your client deployment descriptor.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **View Web Services Client Deployment Descriptor**.

Application-level and server-level bindings are the two levels of bindings that WebSphere Application Server offers. Application-level, server-level, and cell-level are the three levels of bindings that WebSphere Application Server Network Deployment offers. The information in the following implementation descriptions indicates how to configure your application-level bindings. If the Web server is acting as a client, the default bindings are used. To configure the server-level bindings, which are the defaults, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security**.
3. To configure the cell-level bindings, click **Security > Web Services**.

If you are using any of the following configurations, verify that the deployment descriptor is configured properly:
- Request signing
- Request encryption
- BasicAuth authentication
- Identity (ID) Assertion authentication
- Identity (ID) Assertion authentication with the signature TrustMode
- Response digital signature verification
- Response decryption

Request signing

If the integrity constraints (digital signature) are specified, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit > Signing Information > New**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Request encryption

If the confidentiality constraints (encryption) are specified, verify that you configured the encryption information in the binding files.

To configure the encryption parameters, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **> Web Services: Client Security Bindings** .

3. In the Response Receiver Binding column, click **Edit > Encryption Information > New**.

To configure the key locators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

BasicAuth authentication

If BasicAuth authentication is configured as the required security token, specify the CallbackHandler in the binding file to collect the basic authentication data. The following list contains the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
: This implementation prompts for BasicAuth information (user name and password) in an interface.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
: This implementation reads the BasicAuth information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
: This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login binding information, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >***URI_file_name* **> Web Services: Client Security Bindings**.
3. Under Request Sender Bindings, click **Edit > Login Binding**.

Identity (ID) Assertion authentication with BasicAuth TrustMode

Configure a login binding in the bindings file with a com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler implementation. Specify a BasicAuth user ID and password that a TrustedIDEvaluator on a downstream server trusts.

To configure the login binding information, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >***URI_file_name***> Web Services: Client Security Bindings**.
3. Under **Request Sender Bindings**, click **Edit > Login Binding**.

Identity (ID) Assertion authentication with the Signature TrustMode

Configure the signing information in the bindings file with a signing key pointing to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure ID assertion, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings > IDAssertion**.

To configure the login binding information, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >***URI_file_name* **> Web Services: Client Security Bindings**.
3. Under Request Sender Bindings, click **Edit > Login Binding**.

Response digital signature verification

If the integrity constraints (signature required) are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **> Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit > Signing Information > New**.

To configure the trust anchors, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors > New**.

To configure the collection certificate store, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store > New**.

Response decryption

If the confidentiality constraints (encryption) are specified, verify that you defined the encryption information.

To configure the encryption information, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***<URI_file_name* **> Web Services: Client Security Bindings** .
3. In the Response Receiver Binding column, click **Edit > Encryption Information > New**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

View Web services server deployment descriptor

Use this page to view your server deployment descriptor settings.
To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **View Web Services Server Deployment Descriptor**.

WebSphere Application Server has two levels of bindings: application-level and server-level. WebSphere Application Server Network Deployment has three levels of bindings: application-level, server-level, and cell-level. The information in the following implementation descriptions indicate how to configure your application-level bindings. To configure the server-level bindings, which are the defaults, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security**.

To configure the cell-level bindings, click **Security > Web Services**.

- Request digital signature verification
- Request decryption
- BasicAuth authentication
- Identity (ID) Assertion authentication
- Identity (ID) Assertion authentication with the signature TrustMode
- Response signing
- Response encryption

Request digital signature verification

If the integrity constraints (signature required) are defined, verify that you configured the signing information in the binding files.

To configure the signing parameters, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* > **Web Services: Server Security Bindings**.
3. In the Request Receiver Binding column, click **Edit > Signing Information**.

To configure the trust anchor, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

To configure the collection certificate store, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

To configure the key locators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Request decryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To configure the encryption information parameters, complete the following steps:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module**.

3. Under Additional Properties, click **Web Services: Server Security Bindings**. Under Request Receiver Binding, click **Edit > Encryption Information**.

To configure the key locators, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

BasicAuth authentication

If BasicAuth authentication is configured as the required security token, specify the CallbackHandler in the binding file to collect the basic authentication data. The following list contains CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.GuiPromptCallbackHandler
: The implementation prompts for BasicAuth information (user name and password) in an interface panel.

com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
: This implementation reads the BasicAuth information from the binding file.

com.ibm.wsspi.wssecurity.auth.callback.StdPromptCallbackHandler
: This implementation prompts for a user name and password using the standard in (stdin) prompt.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Identity (ID) Assertion authentication with the BasicAuth TrustMode

Configure a login binding in the bindings file with a com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler implementation. Specify a BasicAuth user ID and password that a TrustedIDEvaluator on a downstream server trusts.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Identity (ID) Assertion authentication with the Signature TrustMode

Configure the signing information in the bindings file with a signing key pointing to a key locator. The key locator contains the X.509 certificate that is trusted by the downstream server.

To configure the login mapping information, complete the following steps:
1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

The Java Authentication and Authorization Service (JAAS) uses `WSLogin` as the name of the login configuration. To configure JAAS, click **Security > JAAS Configuration > Application Logins**.

The value of the `<TrustedIDEvaluatorRef>` tag in the binding must match the value of the `<TrustedIDEvaluator>` name.

To configure the trusted ID evaluators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Services, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators**.

Response signing

If the integrity constraints (digital signature) are defined, verify that you have the signing information configured in the binding files.

To specify the signing information, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >***URI_file_name* **> Web Services: Server Security Bindings** .
3. In the Request Receiver Binding column, click **Edit > Signing Information**.

To configure the key locators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Response encryption

If the confidentiality constraints (encryption) are specified, verify that the encryption information is defined.

To specify the encryption information, complete the following steps:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module**.
3. Under Additional Properties, click **Web Services: Server Security Bindings**.
4. Under Request Receiver Binding, click **Edit > Encryption Information**.

To configure the key locators, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Authentication method overview

The Web services security implementation for WebSphere Application Server supports the following authentication methods: BasicAuth, Lightweight Third Party Authentication (LTPA), digital signature, and identity assertion.

When the WebSphere Application Server is configured to use the BasicAuth authentication method, the sender attaches the LTPA token as a BinarySecurityToken from the current security context or from basic authentication data configuration in the binding file in the SOAP message header. The Web services security message receiver authenticates the sender by validating the user name and password against the configured user registry. With the LTPA method, the sender attaches the LTPA BinarySecurityToken it previously received in the SOAP message header. The receiver authenticates the sender by validating the

LTPA token and the token expiration time. With the Digital Signature authentication method, the sender attaches a BinarySecurityToken from a X509 certificate to the Web services security message header along with a digital signature of the message body, time stamp, security token, or any combination of the three. The receiver authenticates the sender by verifying the validity of the X.509 certificate and the digital signature using the public key from the verified certificate.

The identity assertion authentication method is different from the other three authentication methods. This method establishes the security credential of the sender based on the trust relationship. You can use the identity assertion authentication method, for example, when an intermediary server must invoke a service from a downstream server on behalf of the client, but does not have the client authentication information. The intermediary server might establish a trust relationship with the downstream server and then assert the client identity to the same downstream server.

Web Services Security supports the following trust modes:

- BasicAuth
- Digital signature
- Presumed trust

When you use the BasicAuth and digital signature trust modes, the intermediary server passes its own authentication information to the downstream server for authentication. The presumed trust mode establishes a trust relationship using some external mechanism. For example, the intermediary server might pass SOAP messages through a Secure Socket Layers (SSL) connection with the downstream server and transport layer client certificate authentication.

The Web services security implementation for WebSphere Application Server validates the trust relationship by following this procedure:

1. The downstream server validates the authentication information of the intermediary server.
2. The downstream server verifies whether the authenticated intermediary server is authorized for identity assertion. For example, the intermediary server must be in the trust list for the downstream server.

The client identity might be represented by a name string, a distinguished name (DN), or an X.509 certificate. The client identity is attached in the Web services security message in a UsernameToken with just a user name, DN, or in a BinarySecurityToken of a certificate. The following table summarizes the type of security token that is required for each authentication method.

Table 2. Authentication methods and their security tokens

Authentication method	Security token
BasicAuth	BasicAuth requires `<wsse:UsernameToken>` with `<wsse:Username>` and `<wsse:Password>`.
Signature	Signature requires `<ds:Signature>` and `<wsse:BinarySecurityToken>`.

Table 2. Authentication methods and their security tokens (continued)

Authentication method	Security token
IDAssertion	IDAssertion requires <wsse:UsernameToken> with <wsse:Username> or <wsse:BinarySecurityToken> with a X.509 certificate for client identity depending on <idType>. This method also requires other security tokens according to the <trustMode>: • If the <trustMode> is BasicAuth, IDAssertion requires <wsse:UsernameToken> with <wsse:Username> and <wsse:Password>. • If the <trustMode> is Signature, IDAssertion requires <wsse:BinarySecurityToken>.
LTPA	LTPA requires <wsse:BinarySecurityToken> with an LTPA token.

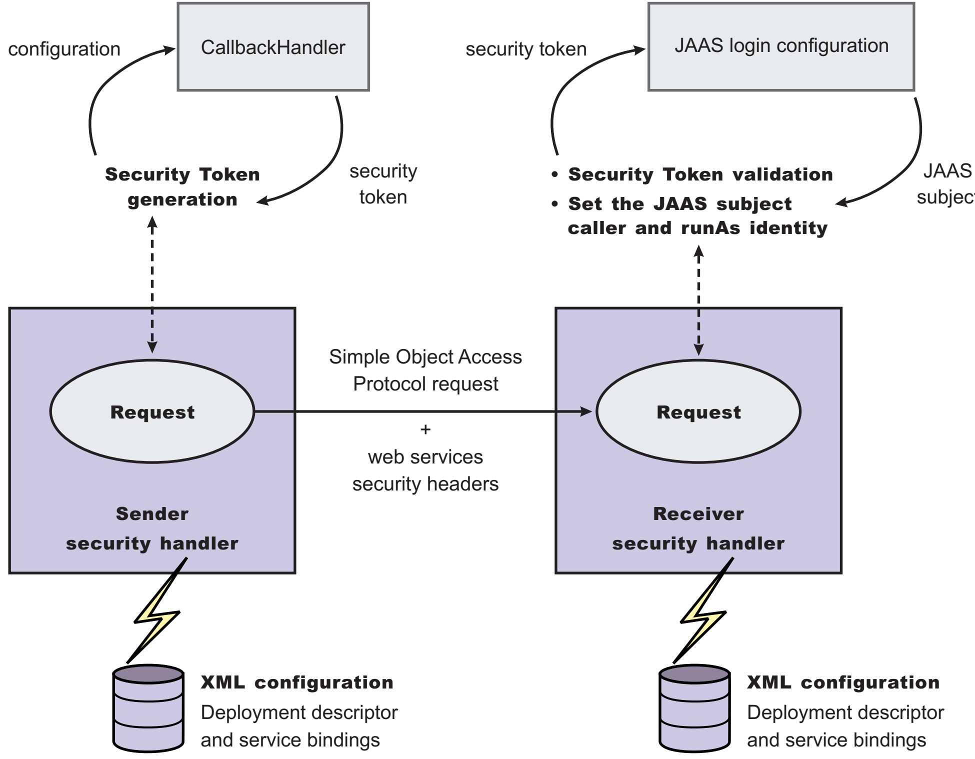
A Web service can support multiple authentication methods simultaneously. The receiver side of the Web services deployment descriptor can specify all the authentication methods that are supported in the `ibm-webservices-ext.xmi` XML file. The Web services receiver-side, as shown in the following example, is configured to accept all the authentication methods described previously:

```
<loginConfig xmi:id="LoginConfig_1052760331326">
      <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
      <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
      <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
      <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

You can define only one authentication method in the sender-side Web services deployment descriptor. A Web service client can use any of the authentication methods that are supported by the particular Web services application. The following example illustrates an identity assertion authentication method configuration in the `ibm-webservicesclient-ext.xmi` deployment descriptor extension of the Web service client:

```
<loginConfig xmi:id="LoginConfig_1051555852697">
      <authMethods xmi:id="AuthMethod_1051555852698" text="IDAssertion"/>
</loginConfig>
<idAssertion xmi:id="IDAssertion_1051555852697" idType="Username" trustMode="Signature"/>
```

As shown in the previous example, the client identity type is `Username` and the trust mode is digital signature.



The sender security handler invokes the `handle()` method of an implementation of the `javax.security.auth.callback.CallbackHandler` interface. The `javax.security.auth.callback.CallbackHandler` interface creates the security token and passes it back to the sender security handler. The sender security handler constructs the security token based on the authentication information in the callback array and inserts the security token into the Web services security message header.

The receiver security handler compares the token type in the message header with the expected token types configured in the deployment descriptor. If none of the expected token types are found in the Web services security header of the SOAP message, the request is rejected with a SOAP fault exception. Otherwise, the token type is used to map to a Java Authentication and Authorization Service (JAAS) login configuration for validating the token. If the authentication is successful, a JAAS Subject is created and associated with the running thread. Otherwise, the request is rejected with a SOAP fault exception.

XML digital signature

XML-Signature Syntax and Processing (XML signature) is a specification that defines XML syntax and processing rules to sign and verify digital signatures for digital content. The specification was developed jointly by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF).

XML signature does not introduce new cryptographic algorithms. WebSphere Application Server uses XML signature with existing algorithms such as RSA, HMAC, and SHA1. XML signature defines many methods for describing key information and enables the definition of a new method.

XML canonicalization (c14n) is often needed when you use XML signature. Information can be represented in various ways within serialized XML documents. For example, although their octet representations are different, the following examples are identical:

- `<person first="John" last="Smith"/>`
- `<person last="Smith" first="John"></person>`

C14n is a process used to canonicalize XML information. Select an appropriate c14n algorithm because the information that is canonicalized is dependent upon this algorithm. One of the major c14n algorithms, Exclusive XML Canonicalization, canonicalizes the character encoding scheme, attribute order, namespace declarations, and so on. The algorithm does not canonicalize white space outside tags, namespace prefixes, or data type representation.

XML signature in the Web Services Security-Core specification

The Web Services Security-Core (WSS-Core) specification defines a standard way for Simple Object Access Protocol (SOAP) messages to incorporate an XML signature. You can use almost all of the XML signature features in WSS-Core except enveloped signature and enveloping signature. However, WSS-Core has some recommendations such as exclusive canonicalization for the c14n algorithm and some additional features such as `SecurityTokenReference` and `KeyIdentifier`.

By including XML signature in SOAP messages, the following are realized:

Message integrity
: A message receiver can confirm that attackers or accidents have not altered parts of the message after these parts are signed by a key.

Authentication
: You can assume that a valid signature is *proof of possession*. A message with a digital certificate issued by a certificate authority and a signature in the message that is validated successfully by a public key in the certificate, is proof that the signer has the corresponding private key. The receiver can authenticate the signer by checking the trustworthiness of the certificate.

XML signature in the current implementation

XML signature is supported in Web services security, however, an application programming interface (API) is not available. The current implementation has many hardcoded behaviors and has some user-operable configuration items. To configure the client for digital signature, see Configuring the client for response digital signature verification: Verifying the message parts. To configure the server for digital signature, see Configuring the server for request digital signature verification: Verifying the message parts.

Security considerations

In a replay attack, an attacker taps the lines, receives a signed message, and then returns the message to the receiver. In this case, the receiver receives the same message twice and might process both of them if the signatures are valid. Processing both messages can cause damage to the receiver if the message is a claim for money. If you have the signed generation time stamp and the signed expiration time in a message replay, attacks might be reduced. However, this is not a complete solution. A message must have a nonce value to prevent these attacks and the receiver must reject a message that contains a processed nonce. The

current implementation does not provide a standard way to generate and check nonces in messages. Applications handle nonces (such as serial numbers) and they need to be signed.

Signing information collection

Use this page to view a list of signing parameters. Signing information is used to sign and validate parts of a message including the body, time stamp, and user name token. You can also use these parameters for X.509 validation when the authentication method is IDAssertion and the ID type is X509Certificate in the server-level configuration. In such cases, you must fill in the certificate path fields only.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* > **Web Services: Server Security Bindings** .
3. In the Request Receiver Binding column, click **Edit > Signing Information**.
4. Click **New** to create a signing parameter. Click **Delete** to delete a signing parameter.

Signature Method:

Specifies the unique name of the signature method.

Signing information configuration settings

Use this page to configure new signing parameters.

The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the W3C document entitled, ″XMLSignature Syntax and Specification: W3C Recommendation 12 Feb 2002″.

To view this administrative console page:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* **> Web Services: Server Security Bindings**.
3. In the Request Receiver Binding column, click **Edit > Signing Information**.
4. Click **New** to create a signing parameter or click **Delete** to delete a signing parameter.

Signature Method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method. This method contains the actual value of the digital signature encoded using base64.

The following algorithms are supported:

- `http://www.w3.org/2000/09/xmldsig#rsa-sha1`
- `http://www.w3.org/2000/09/xmldsig#dsa-sha1`

Digest Method:

Specifies the algorithm URI of the digest method.

The http://www.w3.org/2000/09/xmldsig#sha1 algorithm is supported.

Canonicalization Method:

Specifies the algorithm URI of the canonicalization method.

The following algorithms are supported:
- `http://www.w3.org/2001/10/xml-exc-c14n#`
- `http://www.w3.org/2001/10/xml-exc-c14n#WithComments`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments`

Signing Key:

Specifies the key information that is used for signing. These fields are ignored in receiver-side configuration.

If you specify a **Key Name** and a **Key Locator Reference**, select **None** for the Certificate Path.

Certificate Path:

Specifies the settings for the certificate path validation. When you select **Trust Any**, this validation is skipped and all the incoming certificates are trusted. These fields are ignored in sender-side configuration.

If you click **Trust Any** or select a **Trust Anchor** and a **Certificate Store**, select **None** for the Signing Key in the previous field.

Trust Anchor

The selections available for Trust Anchor are specified by clicking **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Certificate Store

The selections available for the Collection Store are specified by clicking **Servers > Application Servers >***server_name*. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

Signing parameter configuration settings

Use this page to configure new signing parameters.
The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XMLSignature Syntax and Specification: W3C Recommendation 12 Feb 2002*.

To view this administrative console page, complete the following steps:
1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Modules >** *URI_file_name* > **Web Services: Client Security Bindings**.
3. In the Request Sender Binding column, click **Edit > Signing Information**.

If the signing information is not available, select **None**.

If the signing information is available, select **Dedicated Signing Information** and specify the configuration in the following fields:

Signature Method:

Specifies the algorithm Uniform Resource Identifiers (URI) of the signature method. This method contains the actual value of the digital signature encoded using base64.

The following algorithms are supported:
- `http://www.w3.org/2000/09/xmldsig#rsa-sha1`
- `http://www.w3.org/2000/09/xmldsig#dsa-sha1`

Digest Method:

Specifies the algorithm URI of the digest method.

The `http://www.w3.org/2000/09/xmldsig#sha1` algorithm is supported.

Canonicalization Method:

Specifies the algorithm URI of the canonicalization method.

The following algorithms are supported:
- `http://www.w3.org/2001/10/xml-exc-c14n#`
- `http://www.w3.org/2001/10/xml-exc-c14n#WithComments`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`
- `http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments`

Signing Key:

Specifies the key information that is used for signing. These fields are ignored in receiver-side configuration.

If the signing key is not available, select **None**.

Certificate Path:

Specifies the settings for the certificate path validation. When you select **Trust Any**, this validation is skipped and all the incoming certificates are trusted. These fields are ignored in sender-side configuration.

If there is not a certificate path, select **None**.

If there is a certificate path, select **Trust Any** or select a Trust Anchor and a Certificate Store.

Trust Anchor

Specify the selections for the Trust Anchor field by clicking **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Certificate Store

Specify the selections for the Collection Store field by clicking **Servers > Application Servers***server_name*. Under Related Items, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.

Securing Web services using XML digital signature

WebSphere Application Server provides several different methods to secure your Web services; Extensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

XML digital signature provides both message integrity and authentication capabilities when it is used with SOAP messages. A message receiver can verify that attackers or accidents have not altered parts of the message after the message was signed by a key. If a message has a digital certificate issued by a certificate authority (CA) and a signature in the message is validated successfully by a public key in the certificate, it is proof that the signer has the corresponding private key. To use XML digital signature to secure Web services, complete the following steps:

1. Define the security constraints or extensions. To configure the security constraints, you must use the Application Server Toolkit, which is available at the following Web site: http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en
   a. Configure the client to digitally sign a message request. To configure the client, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The client in these steps is the request sender.
      1) Specify the message parts by following the steps found in Configuring the client for request signing: digitally signing message parts.
      2) Select the method used to digitally sign the request message. You can select the digital signature method by following the steps in Configuring the client for request signing: choosing the digital signature method.
   b. Configure the server to verify the digital signature that is used in the message request. To configure the server, you must specify which parts of the SOAP message, sent by the request sender, contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the request receiver, or the server in this step, must match the settings chosen for the request sender in the previous step.
      1) Define the message parts by following the steps found in Configuring the server for request digital signature verification: verifying message parts.
      2) Select the same method used by the request sender to digitally sign the message. You can select the digital signature method by following the steps in Configuring the server for request digital signature verification: choosing the verification method
   c. Configure the server to digitally sign a message response. To configure the server, complete the following steps to specify which parts of the SOAP message to digitally sign and define the method used to digitally sign the message. The sender in these steps is the response sender.

1) Specify which message parts to digitally sign by following the steps found in Configuring the server for response signing: digitally signing message parts.

2) Select the method used to digitally sign the response message. You can select the digital signature method by following the steps in Configuring the server for response signing: choosing the digital signature method

d. Configure the client to verify the digital signature that is used in the message response. To configure the client, you must specify which parts of the SOAP message sent by the response sender contain digitally signed information and which method was used to digitally sign the message. The settings chosen for the response receiver, or client in this step, must match the settings chosen for the response sender in the previous step.

1) Define the message parts by following the steps found in Configuring the client for response digital signature verification: verifying message parts

2) Select the same method used by the response sender to digitally sign the message. You can select the digital signature method by following the steps in Configuring the client for response digital signature verification: choosing the verification method

2. Define the client security bindings. To configure the client security bindings, complete the steps in either of the following topics:
   - Configuring the client security bindings using the Application Server Toolkit
   - Configuring the client security bindings using the administrative console
3. Define the server security bindings. To configure the server security bindings, complete the steps in either of the following topics:
   - Configuring the server security bindings using the Application Server Toolkit
   - Configuring the server security bindings using the administrative console

After completing these steps, you have secured your Web services using XML digital signature.

Transport level security

Transport level security is based on Secure Sockets Layer (SSL) or Transport Layer Security (TLS) that runs beneath HTTP.
SSL and TLS provide security features including authentication, data protection, and cryptographic token support for secure HTTP connections. To run with HTTPS, the service endpoint address must be in the form `https://`.

The integrity and confidentiality of transport data, including SOAP messages and HTTP basic authentication, is confirmed when you use SSL and TLS. See Secure Sockets Layer for more information. Web services applications can also use Federal Information Processing Standard (FIPS) approved ciphers for more secure TLS connections.

WebSphere Application Server uses the Java Secure Sockets Extension (JSSE) package to support SSL and TLS.

HTTP SSL Configuration collection

Use this page to configure transport-level Secure Sockets Layer (SSL) security. You can use this configuration when a Web service is a client to another Web service. You can use transport-level security to enable HTTP SSL (or HTTPS).
Transport-level security can be enabled or disabled independently from

message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the Web service application.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >***URI_file_name* **> Web Services: Client Security Bindings**.
3. Under HTTP SSL Configuration, click **Edit**.

HTTP SSL Enabled:

Specifies secure socket communications for the HTTP transport for this port. When enabled, WebSphere Application Server uses the HTTP SSL Configuration setting.

HTTP SSL Configuration:

Specifies which alias of the SSL configuration to use with the HTTP transport for this port.

This option is used if you select **HTTP SSL Enabled**. SSL aliases are defined in the Secure Sockets Layer configuration repertoire, which you can configure by clicking **Security > SSL**.

HTTP basic authentication

HTTP basic authentication uses a user name and password to authenticate a service client to a secure endpoint.
WebSphere Application Server can have several resources, including Web services, protected by a Java 2 Platform, Enterprise Edition (J2EE) security model.

A simple way to provide authentication data for the service client is to authenticate to the protected service endpoint to the HTTP basic authentication. The basic authentication is located in the HTTP header that carries the SOAP request. When the application server receives the HTTP request, the user name and password are retrieved and verified using the authentication mechanism specific to the server.

Although the basic authentication data is base64-encoded, sending data over HTTPS is recommended. The integrity and confidentiality of the data can be protected by the Secure Sockets Layer (SSL) protocol.

In some cases, a firewall is present using the pass-thru HTTP proxy server. The HTTP proxy server forwards the basic authentication data into the J2EE application server. The proxy server can also be protected. Applications can specify the proxy data by setting properties in a stub object.

HTTP basic authentication collection

Use this page to specify a user ID and password for transport-level basic authentication security for this port. You can use this configuration when a Web service is a client to another Web service.
You can use transport-level security to enable basic authentication. Transport-level security can be enabled or disabled independently from message-level security. Because transport-level security provides minimal security, use message-level security when security is essential to the Web service application.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.

2. Under Related Items, click **Web Module** > *URI_file_name* > **Web Services: Client Security Bindings**.
3. Under HTTP Basic Authentication, click **Edit**.

Basic Authentication ID:

Specifies the user ID for the HTTP basic authentication for this port.

Basic Authentication Password:

Specifies the password for the HTTP basic authentication for this port.

Default configuration for WebSphere Application Server Network Deployment

In the WebSphere Application Server Network Deployment installation, the `ws-security.xml` file is at the cell level and defines the default binding information for Web services security for the entire cell. But each application server can have its own `ws-security.xml` file to override the cell default; similarly, each Web service can override the default in its binding files. The following list contains the defaults defined in `ws-security.xml` file:

Trust anchors
: Identifies the trusted root certificates for signature verification.

Collection certificate stores
: Contains certificate revocation lists (CRLs) and nontrusted certificates for verification.

Key locators
: Locates the keys for digital signature and encryption.

Trusted ID evaluators
: Evaluates the trust of the received identity before identity assertion.

Login mappings
: Contains the Java Authentication and Authorization Service (JAAS) configurations for AuthMethod token validation.

The Web services security run time reads the configuration from the application bindings first, then tries the server-level, and finally tries the cell level. The following figure depicts the run-time configuration process.

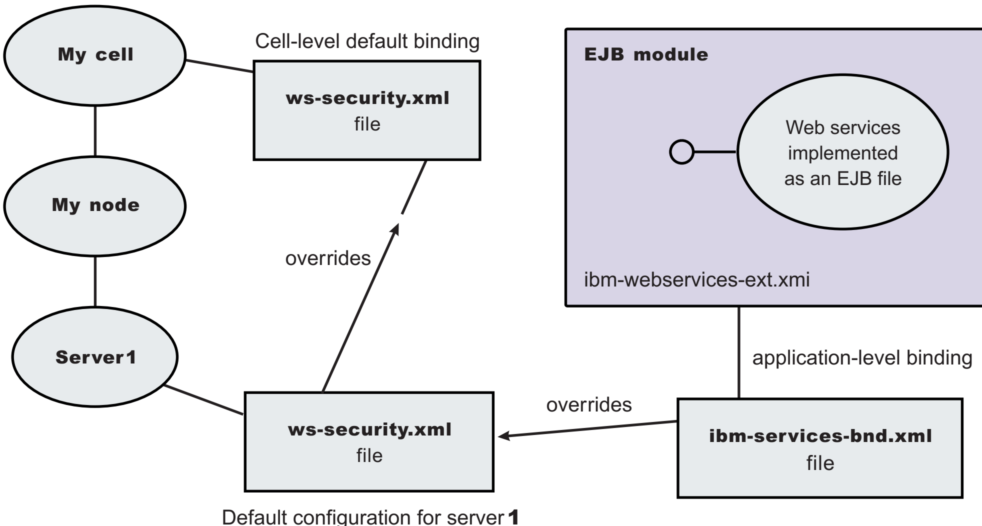


Figure 12. Run-time configuration

Default binding:

The default binding information is defined in the `ws-security.xml` file and can be administered by either the administrative console or by scripting.

Certain applications can share certain binding information. This information includes truststores, keystores, and authentication methods (token validation). WebSphere Application Server provides support for default binding information. Administrators can define binding information at the server level and at the cell level, and applications can refer to this binding information.

You can define the following binding information in the `ws-security.xml` file:

Trust anchors (truststore)

- *Trust anchors* contain key store configuration information that has the root-trusted certificates. Trust anchors are used for certificate path validation of the incoming X.509-formatted security tokens.
- The Trust Anchor Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when Web services is running as a client) to refer to the trust anchor defined in the default binding information. The trust anchor name must be unique in the trust anchor collection.

Collection certificate store

- The *collection certificate store* specifies a list of untrusted, intermediate certificates and is used for certificate path validation of incoming X.509-formatted security tokens. The default provider is IBMCertPath.
- The Certificate Store Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when Web services is running as a client) to refer to the certificate store defined in the default binding information. The Certificate Store Name must be unique to the collection certificate store collection.

Key locators

- *Key locators* specify implementation of the com.ibm.wsspi.wssecurity.config.KeyLocator interface. This interface is used to retrieve keys for signature or encryption. Customer implementations can extend the key locator interface to retrieve keys using other methods. WebSphere Application Server provides implementations to retrieve a key from the key store, map an authenticated identity to a key in the key store, or retrieve a key from the signer certificate (mapping and retrieving actions are used for encrypting the response).
- The Key Locator Name is used in the binding file (`ibm-webservices-bnd.xmi` and `ibm-webservicesclient-bnd-xmi` when Web services is running as a client) to refer to the key locator defined in the default binding information. The Key Locator Name must be unique to the key locators collection in the default binding information.

Trusted ID evaluators

- *Trusted ID evaluators* are an implementation of the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface. This interface is used to make sure the identity (ID)-asserting authority is trusted. Additionally, you can extend the trusted identity evaluator to validate the trust. WebSphere Application Server provides a default implementation for validating trust based on a predefined list of identities.
- The Trusted ID Evaluator Name is used in the binding file (`ibm-webservices-bnd.xmi`) to refer to the trusted identity evaluator defined in the default binding information. The Trusted ID Evaluator Name must be unique to the Trusted ID Evaluator collection.

Login mappings

- *Login mappings* define the mapping of the authentication method to the Java Authentication and Authorization Service (JAAS) login configuration. The mappings are used to authenticate the incoming security token embedded in the Web services security Simple Object Access Protocol (SOAP) message header. The JAAS login configuration is defined in the administrative console under **Security > JAAS Configuration > Application Logins**.
- WebSphere Application Server defines the following authentication methods:

  BasicAuth
  Authenticates user name and password.

  Signature
  Maps the subject distinguished name (DN) in the certificate to a WebSphere Application Server credential.

  IDAssertion
  Maps the identity to a WebSphere Application Server credential.

  LTPA Authenticates a Lightweight Third Party Authentication (LTPA) token.

  After identity authentication, the associated credential is used in the downstream call.
- This method can be extended to authenticate custom security tokens by providing a custom JAAS login configuration and by using the com.ibm.wsspi.wssecurity.auth.module.WSSecurityMappingModule to create the principal and credential required by WebSphere Application Server.
- If LoginConfig (`AuthMethod`) is defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`), but there are no login mapping bindings

(ibm-webservices-bnd.xmi) defined for the AuthMethod, Web services security run time uses the login mapping defined in the default binding information.

WebSphere Application Server

In the WebSphere Application Server, each server has a copy of the ws-security.xml file (default binding information for Web services security). There is no cell-level copy of the ws-security.xml file, which is only available in the WebSphere Application Server Network Deployment installation. To navigate to the server-level default binding in the administrative console, complete the following steps:

1. Click **Servers > Application Servers >** *server1*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**.

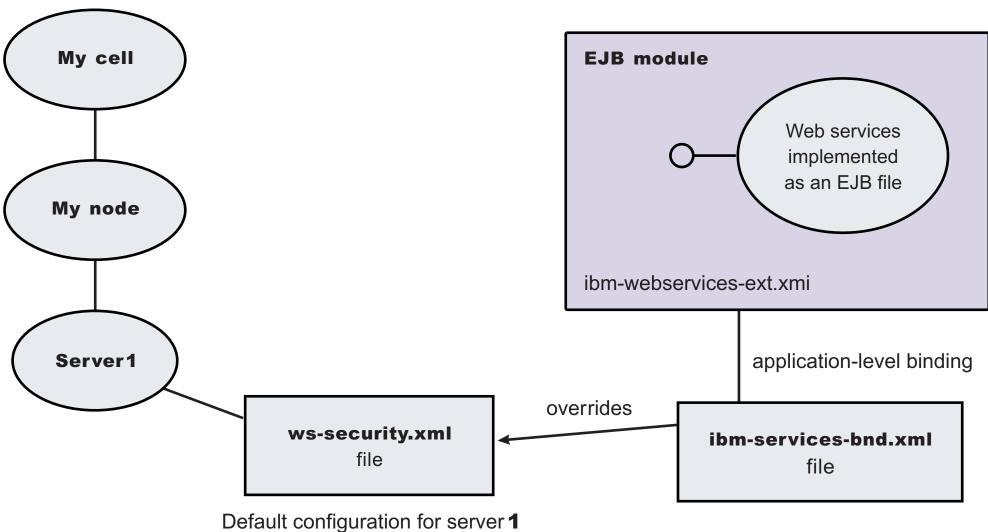


Figure 13. Web services security application-level bindings and server-level default binding information

Web services security run time uses the binding information in the application Enterprise JavaBeans (EJB) or Web module binding file (ibm-webservices-bnd.xmi or ibm-webservicesclient-bnd.xmi if Web services is acting as a client on the server) if the binding information is defined in the application-level binding file. For example, if key locator K1 is defined in both the application-level binding file and the default binding file (ws-security.xml), the K1 in the application-level binding file is used.

WebSphere Application Server Network Deployment

When the WebSphere Application Server is federated to a Network Deployment cell, the default binding file (ws-security.xml) of the server is added to the new cell (with other server level configuration information). If you use the cell-level default binding, the entries of the server level default binding must be removed.

There is a cell-level default binding (ws-security.xml) for Network Deployment installation. Furthermore, for Network Deployment installation server-level binding

is optional. To navigate to the cell-level default binding in the administrative console, click **Security > Web Services**. The server-level binding is described in WebSphere Application Server.

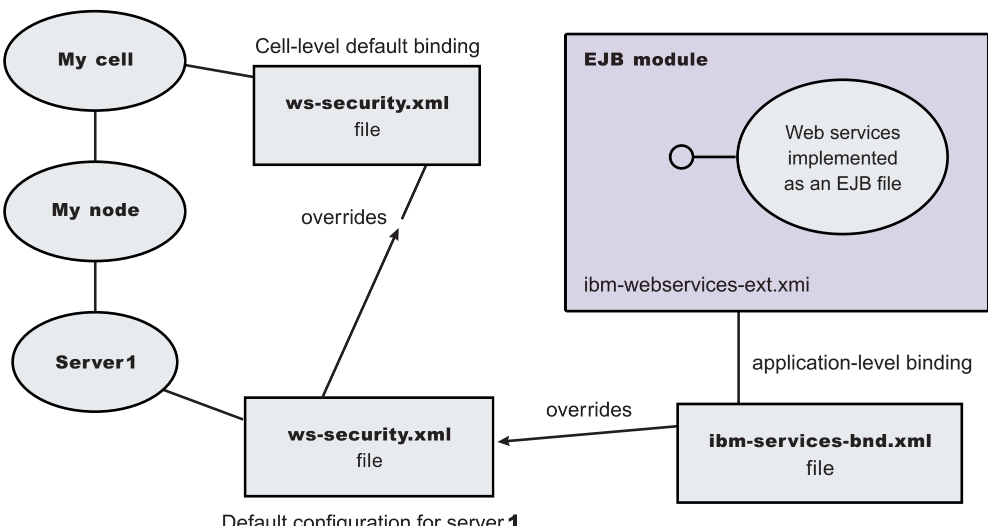


Figure 14. Web services security application-level, cell-level, and server-level default binding information

The order of the default binding information is application-level binding, server-level, and cell-level default binding.

Web services: default bindings for the Web services security collection:

Use this page to configure the settings for nonce on the server level and to manage the default bindings for trust anchors, the collection certificate store, key locators, trusted ID evaluators, and login mappings.

To view this administrative console page, click **Servers > Application Servers** >*server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**.

Read the Web services documentation before you begin defining the default bindings for Web services security.

To define the server bindings, complete the following steps:

1. Click **Applications > Enterprise Applications** >*application_name*.
2. Under Related Items, click **Web Modules** >*URI_file_name* **> Web Services: Server Security Bindings**.

To define the client bindings, complete the following steps:

1. Click **Applications > Enterprise Applications** >*application_name*.
2. Under Related Items, click **Web Modules** >*URI_file_name* **> Web Services: Client Security Bindings**.

The default binding configuration provides a central location where reusable binding information is defined. The application binding file can reference the information contained in the default binding configuration.

Nonce Cache Timeout:

Specifies the timeout value, in seconds, for the nonce cached on the server. Nonce is a randomly generated value.

The Nonce Cache Timeout field is required for the base WebSphere Application Server environment.

If you make changes to the nonce cache timeout value, you must restart WebSphere Application Server for the changes to take effect.

Default	600 seconds
Minimum	300 seconds

Nonce Maximum Age:

Specifies the default time, in seconds, before the nonce time stamp expires. Nonce is a randomly generated value.

The maximum value cannot exceed the number of seconds specified in the Nonce Cache Timeout field for the server level. The value set for this server-level Nonce Maximum Age field must not exceed Nonce Maximum Age value set for the cell level, which you can access by clicking **Security > Web Services > Properties**.

The Nonce Maximum Age field is required for the base WebSphere Application Server environment.

Default	300 seconds
Range	300 to Nonce Cache Timeout seconds

Nonce Clock Skew:

Specifies the default clock skew value, in seconds, to consider when WebSphere Application Server checks the timeliness of the message. Nonce is a randomly generated value.

The maximum value cannot exceed the number of seconds specified in the Nonce Maximum Age field.

The Nonce Clock Skew field is required for the base WebSphere Application Server environment.

Default	0 seconds
Range	0 to Nonce Maximum Age seconds

Trust Anchors:

Specifies a list of keystore objects that contain the trusted root certificates, self-signed or issued by a certificate authority (CA).

The certificate authority authenticates a user and issues a certificate. After the certificate is issued, the keystore objects, which contain these certificates, use the certificate for certificate path or certificate chain validation of incoming X.509-formatted security tokens.

Collection Certificate Store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates. The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key Locators:

Specifies a list of key locator objects that retrieves the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Trusted ID Evaluators:

Specifies a list of trusted ID evaluators that determines whether to trust the identity-asserting authority or the message sender.

The trusted ID evaluators are used to authenticate additional identities from one server to another server. For example, a client sends the identity of user A to server 1 for authentication. Server 1 calls downstream to server 2, asserts the identity of user A, and includes the user ID and password of server 1. Server 2 attempts to establish trust with server 1 by authenticating its user ID and password and checking the trust based on the TrustedIDEvaluator implementation. If the authentication process and the trust check are successful, server 2 trusts that server 1 authenticated user A and a credential is created for user A on server 2 to invoke the request.

Login Mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, use the administrative console and click **Security > JAAS Configuration**.

Web Services: Client security bindings collection:

Use this page to view a list of client-side binding configurations for Web services security. These bindings are used when a Web service is a client to another Web service.

To view this administrative console page, complete the following steps:
1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module >** *URI_file_name* **> Web Services: Client Security Bindings**.

Port:

Specifies the port used to send and receive messages from a server.

Web Service:

Specifies the name of the Web service.

Request Sender Binding:

Specifies the binding configuration used to send request messages to the request receiver.

Click **Edit** to configure the signing information, encryption information, and the login bindings for the request sender and to view a listing of key locators in the key store.

The binding information for the request sender that is specified for the client must match the binding information for the request receiver that is specified for the server.

Response Receiver Binding:

Specifies the binding configuration used to receive response messages from the response sender.

Click **Edit** to configure the signing and encryption information, and to view a list of trust anchors, intermediate certificates found in the collection certificate store, and the key locator objects for the response receiver.

The binding information for the response receiver that is specified for the client must match the binding information for the response sender that is specified for the server.

HTTP Basic Authentication:

Specifies the user ID and password to use for this port with HTTP transport-level basic authentication. You can enable transport-level authentication security independently of message-level security.

Click **Edit** to configure the basic authentication ID and password for transport-level authentication.

HTTP SSL Configuration:

Enables and configures transport-level Secure Socket Layer (SSL) security for this port. You can enable transport-level SSL security independently of message-level security.

Click **Edit** to specify the settings for transport-level HTTP SSL configuration for this port.

Web services: Server security bindings collection:

Use this page to view a list of server-side binding configurations for Web services security.

To view this administrative console page, complete the following steps:

1. Click **Enterprise Applications >** *application_name*.
2. Under Related Items, click **Web Module >** *URI_file_name* **> Web Services: Server Security Bindings**.

Port:

Specifies the port in which messages are received from the request sender.

Web Service:

Specifies the name of the Web service.

Request Receiver Binding:

Specifies the binding configuration used to receive request messages from the request sender.

Click **Edit** to configure the signing information and encryption information and view a listing of trust anchors, intermediate certificates in the collection certificate store, key locators, trusted ID evaluators, and login mappings.

The binding information (for the request receiver) specified for the server must match the binding information (for the request sender) specified for the client.

Response Sender Binding:

Specifies the binding configuration used to send request messages to the response receiver.

Click **Edit** to configure the signing and encryption information, and to view a list of key locator objects for the response sender.

The binding information (for the response sender) specified for the server must match the binding information (for the response receiver) specified for the client.

Trust anchors

A *trust anchor* specifies key stores that contain trusted root certificates that validate the signer certificate.
These key stores are used by the request receiver (as defined in the ibm-webservices-bnd.xmi file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The key stores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these key stores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The trust anchor is defined as javax.security.cert.TrustAnchor in the Java CertPath application programming interface (API). The Java CertPath API uses the trust anchor and the certificate store to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web services security implementation in WebSphere Application Server supports this trust anchor. In WebSphere Application Server, the trust anchor is represented as a Java key store object. The type, path, and password of the key store are passed to the implementation through the administrative console or by scripting.

Configuring trust anchors using the Assembly Toolkit

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level. You can configure an application-level trust anchor using the Assembly Toolkit or the administrative console. This document describes how to configure the application-level trust anchor using the Assembly Toolkit. For more information on creating and configuring trust anchors at the server or cell level, see either Configuring the server security bindings using the Assembly Toolkit or Configuring the server security bindings using the administrative console.

A trust anchor specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The key stores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these key stores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The steps in this document assume that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with Java Specification Requests (JSR) 109 enterprise application. If you have not created a Web services-enabled J2EE with JSR 109 enterprise application, see Developing Web services. Also, see either Configuring the server security bindings using the Assembly Toolkit or Configuring the server security bindings using the administrative console for an introduction on how to manage Web services security binding information on the server.

1. Configure the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` bindings extensions file.
   a. Launch the Assembly Toolkit and click **Windows > Open Perspective > J2EE**.
   b. Select the Web services-enabled Enterprise JavaBeans (EJB) or Web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
   d. Right-click the **`webservicesclient.xml`** file, select **Open With > Web Services Client Editor**, and click the **Web Services Client Binding** tab. The Web Services Client Binding editor is displayed.
   e. Locate the Port Qualified Name Binding section and either select an existing entry or click **Add**, to add a new port binding. The Web Services Client Port Binding editor displays for the selected port.
   f. Locate the Trust Anchor section and click **Add**. The Trust Anchor dialog box is displayed.

1) Enter a unique name within the port binding for the **Trust anchor name**.

   The name is used to reference the trust anchor that is defined.

2) Enter the key store password, path, and key store type.

   The supported key store types are Java Cryptography Extension (JCE) and JCEKS.

Click **Edit** to edit the selected trust anchor.

Click **Remove** to remove the selected trust anchor.

When you start the application, the configuration is validated in the run time while the binding information is loading.

g. Save the changes.

2. Configure the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` bindings extensions file.
   a. Launch the Assembly Toolkit and click **Windows > Open Perspective > J2EE**.
   b. Select the Web services enabled EJB or Web module.
   c. In the Package Explorer window, click the `META-INF` directory for an EJB module or the `WEB-INF` directory for a Web module.
   d. Right-click the **`webservices.xml`** file, select **Open With > Web Services Editor**, and click the **Bindings** tab. The Web Services Bindings editor is displayed.
   e. Locate the Web Service Description Bindings section and either select an existing entry or click **Add** to add a new Web services descriptor.
   f. Click **Binding Configurations**. The Web Services Binding Configurations editor is displayed for the selected Web services descriptor.
   g. Locate the Trust Anchor section and click **Add**. The Trust Anchor dialog box is displayed.
      1) Enter a unique name within the binding for the **Trust anchor name**.

         This unique name is used to reference the trust anchor defined.

      2) Enter the key store password, path, and key store type. The supported key store types are JCE and JCEKS.

      Click **Edit** to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor.

      When you start the application, the configuration is validated in the run time while the binding information is loading.

   h. Save the changes.

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the Web services is acting as client) to verify the signer certificate.

The request receiver or the response receiver (if the Web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

To complete the signing information configuration process for request receiver, complete the following tasks:

1. Configure the server for request digital signature verification: Verifying the message parts
2. Configure the server for request digital signature verification: Choosing the verification method

To complete the process for the response receiver, if the Web services is acting as a client, complete the following tasks:
1. Configure the client for response digital signature verification: Verifying the message parts
2. Configure the client for response digital signature verification: Choosing the verification method

Configuring trust anchors using the administrative console

This document describes how to configure trust anchors or trust stores at the application level. It does not describe how to configure trust anchors at the server or cell level. For more information on creating and configuring trust anchors at the server or cell level, see either Configuring the server security bindings using the Application Server Toolkit or Configuring the server security bindings using the administrative console.

Important: Trust anchors defined at the application level have a higher precedence over trust anchors defined at the server or cell level.

You can configure an application-level trust anchor using the Application Server Toolkit or the administrative console. This document describes how to configure the application-level trust anchor using the administrative console.

A trust anchor specifies key stores that contain trusted root certificates, which validate the signer certificate. These key stores are used by the request receiver (as defined in the `ibm-webservices-bnd.xmi` file) and the response receiver (as defined in the `ibm-webservicesclient-bnd.xmi` file when Web services is acting as client) to validate the signer certificate of the digital signature. The keystores are critical to the integrity of the digital signature validation. If they are tampered with, the result of the digital signature verification is doubtful and comprised. Therefore, it is recommended that you secure these keystores. The binding configuration specified for the request receiver in the `ibm-webservices-bnd.xmi` file must match the binding configuration for the response receiver in the `ibm-webservicesclient-bnd.xmi` file.

The steps in this document assume that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with Java Specification Requests (JSR) 109 enterprise application. If not you have not created a Web services-enabled J2EE with JSR 109 enterprise application, see Developing Web services. Also, see either Configuring the server security bindings using the Application Server Toolkit or Configuring the server security bindings using the administrative console for an introduction on how to manage server Web services security binding information.

Important: Before completing the following steps, it is assumed that a Web services-enabled enterprise application was deployed to the WebSphere Application Server.

The following steps are for the client-side response receiver, which is defined in the `ibm-webservicesclient-bnd.xmi` file and the server-side request receiver, which is defined in the `ibm-webservices-bnd.xmi` file.

1. Click **Applications > Enterprise Applications >** *enterprise_application*.
2. In the Related Links section, click either **EJB Modules** or **Web Modules** and then click the Web services-enabled module in the **Uri** field.
3. Under Additional Properties, click **Web Services: Client Security Bindings** to edit the response receiver binding information, if Web services is acting as client.
   a. Under Response Receiver Binding, click **Edit**.
   b. Under Additional Properties, click **Trust Anchors**.
   c. Click **New** to create a new trust anchor.

      Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.

      Enter the key store password, path, and key store type.

      Click the trust anchor name link to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor or anchors.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
4. Return to Web services-enabled module panel accessed in step 2.
5. Under Additional Properties, click **Web Services: Server Security Bindings** to edit the request receiver binding information.
   a. Under Request Receiver Binding, click **Edit**.
   b. Under Additional Properties, click **Trust Anchors**.
   c. Click **New** to create a new trust anchor

      Enter a unique name within the request receiver binding for the **Trust anchor name** field. The name is used to reference the trust anchor that is defined.

      Enter the key store password, path, and key store type.

      Click the trust anchor name link to edit the selected trust anchor.

      Click **Remove** to remove the selected trust anchor or anchors.

      When you start the application, the configuration is validated in the run time while the binding information is loading.
6. Save the changes.

This procedure defines trust anchors that can be used by the request receiver or the response receiver (if the Web services is acting as client) to verify the signer certificate.

The request receiver or the response receiver (if the Web service is acting as a client) uses the defined trust anchor to verify the signer certificate. The trust anchor is referenced using the trust anchor name.

To complete the signing information configuration process for request receiver, complete the following tasks:

1. Configuring the server for request digital signature verification: verifying the message parts
2. Configuring the server for request digital signature verification: choosing the verification method

To complete the process for the response receiver, if the Web services is acting as client, complete the following tasks:
1. Configuring the client for response digital signature verification: verifying the message parts
2. Configuring the client for response digital signature verification: choosing the verification method

Trust anchors collection:

Use this page to view a list of keystore objects that contain trusted root certificates. These objects are used for certificate path validation of incoming X.509-formatted security tokens. Keystore objects within trust anchors contain trusted root certificates used by the CertPath API to validate the trust of a certificate chain.

To create the keystore file, use the key tool located in the `*install_dir*\java\jre\bin\keytool` directory.

To view this administrative console page, click **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors**.

Click **New** to create a new trust anchor.

Click **Delete** to a delete trust anchor.

If you click **Update runtime**, the Web services security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. If you make changes on this panel, you must complete the following steps:
1. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you are returned to the administrative console home panel.
2. Return to the Trust Anchors collection panel and click **Update runtime**.

   Important: When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Trust Anchor Name:

Specifies the unique name used to identify the trust anchor.

Key Store Path:

Specifies the location of the keystore file that contains the trust anchors.

Key Store Type:

Specifies the type of keystore file.

The value for this field is either **JKS** or **JCEKS**.

Trust anchor configuration settings:

Use this information to configure a trust anchor. Trust anchors point to key stores that contain trusted root or self-signed certificates. This information enables you to specify a name for the trust anchor and the information needed to access a key store. The application binding uses this name to reference a predefined trust anchor definition in the binding file (or the default).

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust Anchors > New**.

Trust Anchor Name:

Specifies the unique name used by the application binding to reference a predefined trust anchor definition in the default binding.

Key Store Password:

Specifies the password needed to access the keystore file.

Key Store Path:

Specifies the location of the keystore file.

Use ${USER_INSTALL_ROOT} as this path expands to the WebSphere Application Server path on your machine.

Key Store Type:

Specifies the type of key store file.

The value in this field is either **JKS** or **JCEKS**.

JKS
: Specify this option if you are not using Java Cryptography Extensions (JCE).

JCEKS
: Specify this option if you are using Java Cryptography Extensions. Although the JCEKS key store format is more secure, it decreases performance.

Data type	String
Default	JKS
Range	JKS, JCEKS

Collection certificate store

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. The collection certificate stores are utilized when processing a received SOAP message. They are configured in the securityRequestReceiverBindingConfig section of the binding file for servers and in the securityResponseReceiverBindingConfig section of the binding file for clients.

A collection certificate store is one kind of certificate store. A certificate store is defined as `javax.security.cert.CertStore` in the Java CertPath Application Programming Interface (API). The Java CertPath API defines the following types of certificate stores:

Collection certificate store
: A collection certificate store accepts the certificates and CRLs as Java collection objects.

Lightweight Directory Access Protocol (LDAP) certificate store
: The LDAP certificate store accepts certificates and CRLs as LDAP entries.

The CertPath API uses the certificate store and the trust anchor to validate the incoming X.509 certificate that is embedded in the SOAP message.

The Web services security implementation in the WebSphere Application Server supports the collection certificate store. Each certificate and CRL is passed as a encoded file. This configuration is done using either the administration console or by scripting.

Configuring the client-side collection certificate store using the Application Server Toolkit

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the WebSphere Application Server Toolkit.

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab in the Web Services Client Editor within the WebSphere Application Server Toolkit. The Web Services Client Port Binding window displays.
6. Select one of the Port Qualified Name Binding entries.
7. Expand the **Security Response Receiver Binding Configuration > Certificate Store List > Collection Certificate Store** section.
8. Click **Add** to create a new collection certificate store, **Edit** to edit an existing certificate store, or **Remove** to delete an existing certificate store.
9. Enter a name in the **Name** field. This is a name that is referenced in the **Certificate store reference** field in the Signing info dialog box.
10. Leave the **Provider** field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be:`${`*`USER_INSTALL_ROOT`*`}/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you are done adding paths.

Configuring the client-side collection certificate store using the administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the client-side collection certificate store using the administrative console.

1. Connect to the WebSphere Application Server administrative console. You can connect to the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click either **Web Services: Client Security Bindings** to add the collection certificate store to the client security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the security extensions for either the client or the server.

   To configure the security extensions for the client, see the following topics:
   - Configuring the client for response digital signature verification: verifying the message parts
   - Configuring the client for response digital signature verification: choosing the verification method
6. Click **Edit** under Response Receiver Binding to edit the client security bindings.
7. Click **Collection Certificate Store**.
8. Click a Certificate Store Name to edit an existing certificate store or click **New** to add a new certificate store name.
9. Enter a name in the **Certificate Store Name** field. The name entered in this field is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
10. Leave the **Certificate Store Provider** field as `IBMCertPath`.
11. Click **Apply**.
12. Under Additional Properties, click **X.509 Certificates > New**.
13. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.
14. Click **OK**.

Configuring the server-side collection certificate store using the Assembly Toolkit

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the Assembly Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using the Assembly Toolkit.

1. Launch the Assembly Toolkit and click**Windows > Open Prospective > J2EE**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit. The Web Service Binding Configuration window displays.
6. Select one of the Web service description binding entries under the Port Component Binding section.
7. Expand the **Request Receiver Binding Configuration Details > Certificate Store List > Collection Certificate Store** section.
8. Click **Add** to create a new collection certificate store, **Edit** to edit an existing certificate store, or **Remove** to delete an existing certification store.
9. Enter a name in the **Name** field. This is a name that is referenced in the **Certificate store reference** field in the Signing info dialog.
10. Leave the **Provider** field as `IBMCertPath`.
11. Click **Add** to enter the path to your certificate store. For example, the path might be:`${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have additional certificate store paths, click **Add** to add the paths.
12. Click **OK** when you are done adding paths.

Collection certificate store collection:

Use this page to view a list of certificate stores containing untrusted, intermediary certificate files awaiting validation. Validation might consist of checking to see if the certificate is on a certificate revocation list (CRL), checking that the certificate has not expired, and checking that the certificate was issued by a trusted signer.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers** *server_name*.
2. Under **Related Items**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
3. Click **New** to specify a store name and provider for a new collection certificate store.
4. Click **Delete** to delete a collection certificate store.

Using this panel, complete the following steps:

1. Specify a certificate store name and certificate store provider.
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you are returned to the administrative console home panel.
3. Return to the collection certificate store collection panel and click **Update runtime**to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.

Important: When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Certificate Store Name:

Specifies the name of the certificate store.

Certificate Store Provider:

Specifies the provider of the certificate store.

Collection certificate store configuration settings:

Use this page to specify the name and provider of a certificate store.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >***server_name*.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store > New**.

Certificate Store Name:

Specifies the name for the certificate store. The application binding uses the certificate store name to reference a predefined binding.

Certificate Store Provider:

Specifies the provider for the certificate store implementation.

Data type	String
Default	IBM CertPath

X.509 certificates collection:

Use this page to view a list of X.509 certificates.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
3. On the Collection Certificate Store page under Additional Properties, click **X.509 Certificates**.

Click **New** to create a new path to an X.509 certificate.

Click **Delete** to delete a path to an X.509 certificate.

X509 Certificate Path:

Specifies the location of the X.509 certificate.

X.509 certificate configuration settings:

Use this page to specify the location of your X.509 certificates.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >***server_name*.
2. Under **Additional Properties**, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
3. On the *Collection Certificate Store* page, under **Additional Properties**, click **X.509 Certificates > New**.

X509 Certificate Path:

Specifies the location of the X.509 certificate.

Configuring the server-side collection certificate store using the administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message.

You can configure the collection certificate either by using the WebSphere Application Server Toolkit or the WebSphere Application Server administrative console. Complete the following steps to configure the server-side collection certificate store using the administrative console.

1. Connect to the WebSphere Application Server administrative console. You can connect to the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click **Web Services: Server Security Bindings** to add the collection certificate store to the server security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the security extensions for the server.

   To configure the security extensions for the server, see the following topics:

   - Configuring the server for request digital signature verification: verifying the message parts
   - Configuring the server for request digital signature verification: choosing the verification method
6. Click **Edit** under Request Receiver Binding to edit the server security bindings.
7. Click **Collection Certificate Store**.
8. Click a Certificate Store Name to edit an existing certificate store or click **New** to add a new certificate store name.
9. Enter a name in the **Certificate Store Name** field. The name entered in this field is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
10. Leave the **Certificate Store Provider** field as `IBMCertPath`.
11. Click **Apply**.
12. Under Additional Properties, click **X.509 Certificates > New**.
13. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`. If you have any additional certificate store paths to enter, click **New** and add the path names.

14. Click **OK**.

Configuring default collection certificate stores at the server level in the WebSphere Application Server administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates. You also can specify the default binding information at the cell level.

Complete the following steps to configure the default collection certificate store at the server level using the WebSphere Application Server administration console:

1. Connect to administrative console. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Servers > Application Servers >** *server1*.
3. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Collection Certificate Store**.
4. Enter a name in the **Certificate Store Name** field. This is a name that is referenced in the Certificate Store field on the Signing information configuration page.
5. Leave the **Certificate Store Provider** field as `IBMCertPath`.
6. Click **Apply**.
7. Under Additional Properties, click **X.509 Certificates > New**.
8. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`.

   If you have any additional certificate store paths to enter, click **New** and add the path names.
9. Click **OK**.

Configuring default collection certificate stores at the cell level in the WebSphere Application Server administrative console

A collection certificate store is a collection of non-root, certificate authority (CA) certificates and certificate revocation lists (CRLs). This collection of CA certificates and CRLs are used to check the signature of a digitally signed SOAP message. A certificate store typically refers to a certificate store located in the file system. The location of the certificate store can vary from machine to machine so you might configure a default collection certificate store for a specific machine and reference it from within the signing information. The signing information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single collection certificate store for all of the applications that need to use the same certificates. You also can specify the default binding information at the server level.

Complete the following steps to configure the default collection certificate store at the cell level using the WebSphere Application Server administration console:

1. Connect to administrative console. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Security > Web Services > Collection Certificate Store**.
3. Click a listed Certificate Store Name to edit an existing store, or click **New** to add a new store. This is a name that is referenced in the **Certificate Store** field on the Signing information configuration page.
4. Leave the **Certificate Store Provider** field as `IBMCertPath`.
5. Click **Apply**.
6. Under Additional Properties, click **X.509 Certificates > New**.
7. Enter the path to your certificate store. For example, the path might be: `${USER_INSTALL_ROOT}/etc/ws-security/samples/intca2.cer`

   If you have any additional certificate store paths to enter, click **New** and add the path names.
8. Click **OK**.

Key locator

A key locator (`com.ibm.wsspi.wssecurity.config.KeyLocator`) is a abstraction of the mechanism that retrieves the key for digital signature and encryption.
You can use any of the following infrastructure from which to retrieve the keys depending upon the implementation:

- Java keystore file
- Database
- LDAP server

Key locators search the key using some type of a clue. The following types of clues are allowed:

- A string label of the key, which is explicitly passed through the application programming interface (API). The relationships between each key and its name (string label) is maintained inside the key locator.
- The execution context of the key locator; explicit information is not passed to the key locator. A key locators, by itself, determines the appropriate key according to their execution context.

For example, key locators can obtain the identity of the caller from the context and can retrieve the public key of the caller for response encryption.

Restriction: Current versions of key locators do not support the retrieval of verification keys because current Web services security implementations do not support the secret key-based signature. Since the key locators support the public key-based signature only, the key for verification is embedded in the X.509 certificate as a `<BinarySecurityToken>` element in the incoming message.

Usage scenarios

This section describes the usage scenarios for key locators.

Signing

The name of the signing key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned. The corresponding X.509 certificate also can be returned.

Verification

As described previously, key locators are not used in signature verification.

Encryption

The name of the encryption key is specified in the Web services security configuration. This value is passed to the key locator and the actual key is returned.

Decryption

The Web services security specification recommends the usage of the key identifier instead of the key name. However, while the algorithm for computing the identifier for the public keys is defined in Internet Engineering Task Force (IETF) Request for Comment (RFC) 3280, there is no agreed upon algorithm for the secret keys. Therefore, the current implementation of Web services security uses the identifier only when public key-based encryption is performed. Otherwise, the ordinal key name is used.

When you use public key-based encryption, the value of key identifier is embedded in the incoming encrypted message. Then, the Web services security implementation searches for all the keys managed by the key locator and decrypts the message using the key whose identifier value matches the one in the message.

When you use secret key-based encryption, the value of key name is embedded in the incoming encrypted message. The Web services security implementation asks the key locator for the key whose name matches the one in the message and decrypts the message using the key.

Key locator collection:

Use this page to view a list of available key locators. Key locators identify the keys needed for digital signature and encryption. A key locator must implement the com.ibm.wsspi.wssecurity.config.KeyLocator interface. The two default implementations are:
com.ibm.wsspi.wssecurity.config.WSldKeyStoreMapKeyLocator and
com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.
3. Click **New** to create a key locator. Click **Delete** to delete a key locator.

Using this panel, complete the following steps:
1. Specify a key locator name and key locator class name on the panel
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.

3. After saving your changes, return to the **Key Locator** collection panel to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.
4. To update the Web services security run time, click **Update runtime**. When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.
5. Once you define key locators, click the key locator name to specify additional properties and keys under **Additional Properties**.

Key Locator Name:

Specifies the unique name of the key locator.

Key Locator Classname:

Specifies the class name of the key locator in the keystore file.

Key locator configuration settings:

Use this page to specify the settings for key locators.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators > New**.

Key Locator Name:

Specifies the name of the key locator.

Data type	String

Key Locator Classname:

Specifies the name for the key locator class implementation.

WebSphere Application Server has the following default key locator class implementations:

com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
: This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, user1 is mapped to `mappedName_1`. Otherwise, name="default". When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
: This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. The encryption process uses this class to obtain a key to encrypt a message and the digital signature process uses this class to obtain a key to sign a message. The

com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator class maps a logical name to a key alias in the keystore file. For example, key #105115176771 is mapped to `CN=Alice, O=IBM, c=US`.

Data type	String	
Defaults	com.ibm.wsspi .wssecurity.config .KeyStoreKeyLocator	com.ibm.wsspi .wssecurity.config .WSldKeyStoreMayKeyLocator

Key Store Password:

Specifies the password used to access the keystore file.

Key Store Path:

Specifies the location of the keystore file.

Use ${USER_INSTALL_ROOT} as this path expands to the WebSphere Application Server path on your machine.

Key Store Type:

Specifies the type of the keystore file.

The value for this field is either **JKS** or **JCEKS**:

JKS Use this option if you are not using Java Cryptography Extensions (JCE).

JCEKS
Use this option if you are using Java Cryptography Extensions.

Default	JKS
Range	JKS, JCEKS

Keys

Keys are used for XML signature and encryption.
Largely, there two kinds of keys are used in the current Web services security implementation:

- Public key - such as RSA and DSA
- Secret key - such as DES

In public-key-based signature, a message is signed using sender private key and is verified using the sender public key. In public key-based encryption, a message is encrypted using receiver public key and is decrypted using the receiver private key. In secret key-based signature and encryption, the same key is used by both parties.

While the current implementation of Web services security can deal with both kinds of keys, there are a few things to be noted:

- Secret key-based signature is not supported.
- The format of the message differ slightly between public key-based encryption and secret key-based encryption.

Key collection:

Use this page to view a list of logical names that are mapped to a key alias in the keystore file.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators >** *key_locator_name*.
3. Under Additional Properties, click **Keys**.
4. Click **New** to create a new key object in the keystore file.
5. Click **Delete** to a delete a mapping of a key object within the keystore file.

Key Name:

Specifies the name of the key object found in the keystore file.

Key Alias:

Specifies an alias for the key object.

The alias is used when the key locator searches for the key objects in the keystore.

Key configuration settings:

Use this page to define a mapping of a logical name to a key alias in a keystore file.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators >** *key_locator_name*.
3. Under Additional Properties, click **Keys > New**.

Key Name:

Specifies the name of the key object. This name is used by the key locator to find the key within the keystore file.

Key Alias:

Specifies the alias for the key object contained in the keystore file.

Key Password:

Specifies the password needed to access the key object within the keystore file.

Web services security service provider programming interfaces

Several Service Provider Programming Interfaces (SPIs) are provided to extend the capability of the Web services security run time. The following is a list of SPIs that are available for WebSphere Application Server:
- `com.ibm.wsspi.wssecurity.config.KeyLocator` is an abstract for obtaining the keys for digital signature and encryption. The following are the default implementations:
  - `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`

Implements the Java keystore.

  - `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator`

    Provides a mapping of authenticated identity to a key for encryption or use the default key specified. This is typically used in the response sender configuration.
  - `com.ibm.wsspi.wssecurity.config.CertInRequestKeyLocator`

    Provides the capability of using the signer key for encryption in the response message. This is typically used in the response sender configuration.
- `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` is an interface that used to evaluate the trust for identity assertion. The following is the default implementation:
  - `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`

    Enables you to define a list of trusted identities.
- The JAAS CallbackHandler Application Programming Interfaces (APIs) are used for token generation by the request sender. This can be extended to generate custom token to be inserted in the Web services security header. The following are the default implementations are provided by WebSphere Application Server:
  - `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`

    Presents a login prompt to gather the basic authentication data. Use this implementation in the client environment only.
  - `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`

    Collects the basic authentication data in the standard in (stdin) prompt. Use this implementation in the client environment only.
  - `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

    Reads the basic authentication data from the application binding file. This might be used on the server side to generate a user name token.
  - `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

    Generates Lightweight Third Party Authentication (LTPA) token in the Web services security header as binary security token. If there is basic authentication data defined in the application binding file, it is used to perform a login, extract the LTPA token from the WebSphere credentials, and insert the token in the Web services security header. Otherwise, it will extract the LTPA security token from the invocation credentials (run as identity) and insert the token in the Web services security header.

The JAAS LoginModule API is used for token validation on the request receiver side of the message. You can implement a custom LoginModule to perform validation of the custom token on the request receiver of the message. Once the token is verified and validated, the token is set as the caller, run as identity in the WebSphere run time, and the identity is used for authorization checks by the containers before a Java 2 Platform, Enterprise Edition (J2EE) resource is invoked. The following are the default "AuthMethod" configurations provided by WebSphere Application Server:

BasicAuth
: Validates a user name token

Signature
: Maps a distinguished name (DN) of a verified certificate to a Java Authentication and Authorization Service (JAAS) subject.

IDAssertion
: Maps a trusted identity to a JAAS subject.

LTPA Validates an LTPA token received in the message and creates a JAAS subject.

Configuring key locators using the Assembly Toolkit

This task provides instructions on how to configure key locators using the Assembly Toolkit. You can configure key locators in various locations within the Assembly Toolkit. This task provides instructions how to configure key locators at any of these locations because the concept is the same.

1. Launch the Assembly Toolkit and click **Windows > Open Prospective > J2EE**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file and click **Open With > Web Services Client Editor** or right-click the `webservices.xml` file and click **Open With > Web Services Editor**.
5. Click the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit or the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit.
6. Expand one of the Binding Configuration sections.
7. Expand the Key Locators section.
8. Click **Add**to create a new key locator, **Edit** to edit an existing key locator, or **Remove** to delete an existing key locator.
9. Enter a key locator name. The name entered for the **Key locator name** is used to refer to the key locator from the Encryption information and Signing Information sections.
10. Enter a key locator class. The key locator class is the implementation of the `KeyLocator` interface. When using default implementations, select a class from the menu
11. Determine whether to click **Use key store**. Select this option when you use the default implementations as they use key stores. If you click **Use key store**, complete the following steps:
    a. Enter a value in the **key store storepass** field. The key store storepass is the password used to access the key store.
    b. Enter a path name in the **key store path** field. The key store path is the location on the file system where the key store resides. Make sure that the location can be found wherever you deploy the application.
    c. Enter a type value in the **key store type** field. The valid types to enter are JKS and JCEKS. JKS is used when you are not using Java Cryptography Extensions (JCE). JCEKS is used when you are using JCE. Although the JCEKS type is more secure, it might decrease performance.
    d. Click **Add** to create an entry for a key in the key store.
        1) Enter a value in the **Alias** field.

           The key alias is a reference to this particular key from the Signing Information section.
        2) Enter a value in the **Key pass** field.

           The key pass is the password associated with the certificate when created using Java Development Kit, `keytool.exe`.
        3) Enter a value in the **Key name** field.

           The key name refers to the alias of the certificate as found in the key store.

12. Click **Add** to create a custom property. The property can be used by custom implementations of `KeyLocator`. For example, you can use properties with the `WSIdKeyStoreMapKeyLocator` default implementation. The `KeyLocator` has the following property names:
   - *id_*, which maps to a credential user ID.
   - *mappedName_* , which maps to the key alias to use for this user name.
   - *default*, which maps to a key alias to use when a credential does not have an associated *id_* entry

   A typical set of properties for this key locator might be: `id_1=user1`, `mappedName_1=key1`, `id_2=user2`, `mappedName_2=key2`, `default=key3`. If `user1` or `user2` authenticates, then the associated `key1` or `key2` is used, respectively. However, if none of the user properties authenticate or the user is not `user1` or `user2`, then `key3` is used.
   a. Enter a name in the **Name** field. The name entered is the property name.
   b. Enter a value in the **Value** field. This value entered is the property value.

Configuring key locators using the administrative console

This task provides instructions on how to configure key locators using the WebSphere Application Server administrative console. You can configure binding information in the administrative console, but for extensions, you must use the Application Server Toolkit. The following steps are used to configure a key locator in the administrative console for a specific application:

1. Connect to administrative console by typing `http://:9090/admin` in your Web browser unless you have changed the port number.
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules**, depending on the type of module you are securing.
4. Click the name of the module you are securing.
5. Under Additional Properties, click either **Web Services: Client Security Bindings** or **Web Services: Server Security Bindings** depending on whether you are adding the key locator to the client security bindings or the server security bindings. If you do not see any entries, return to the WebSphere Application Server Toolkit and configure the Security Extensions.
6. Edit the Request Sender Binding, Response Receiver Binding, Request Receiver Binding, or Response Sender Binding
   - If you are editing your client security bindings, click **Edit** for either the Request Sender Binding or Response Receiver Binding.
   - If you are editing your server security bindings, click **Edit** for either the Request Receiver Binding or Response Sender Binding.
7. Click **Key Locators**.
8. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
   a. Specify a name for the key locator in the **Key Locator Name** field.
   b. Specify a name for the key locator class implementation in the **Key Locator Classname** field. WebSphere Application Server has the following default key locator class implementations:

com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator
: This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
: This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice`, `O=IBM`, `c=US`.

c. Specify the password used to access the keystore password in the **Key Store Password** field. This field is optional is the key locator does not use a keystore.
d. Specify the path name used to access the keystore in the **Key Store Path** field. This field is optional is the key locator does not use a keystore. Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.
e. Select a keystore type from the **Key Store Type** field. This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.

Configuring server and cell level key locators using the administrative console

A key locator typically locates a key store in the file system. The location of key stores can vary from machine to machine so it is often helpful to configure a default key locator for a specific machine and reference it from within the encryption or signing information. This information is found within the binding configurations of any application installed on the machine. This suggestion enables you to define a single key locator for all applications that need to use the same keys. In a Network Deployment environment, you also can specify the default binding information at the cell level.

This task provides instructions on how to configure server and cell-level key locators for a specific application using the WebSphere Application Server administrative console. You can configure binding information in the administrative console, but for extensions, you must use the Application Server Toolkit.

- Configure default key locators at the server level
  1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
  2. Click **Servers > Application Servers >***server1*.

3. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.
4. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:
   a. Specify a name for the key locator in the **Key Locator Name** field.
   b. Specify a name for the key locator class implementation in the **Key Locator Classname** field.

      WebSphere Application Server has the following default key locator class implementations:

      `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator`
      This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The `com.ibm.wsspi.wssecurity.config.WSIdKeyStoreMapKeyLocator` class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, `user1` is mapped to `mappedName_1`. Otherwise, `name="default"`. When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

      `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator`
      This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The `com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator` class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to `CN=Alice`, `O=IBM`, `c=US`.
   c. Specify the password used to access the keystore password in the **Key Store Password** field.

      This field is optional is the key locator does not use a keystore.
   d. Specify the path name used to access the keystore in the **Key Store Path** field.

      This field is optional is the key locator does not use a keystore. Use `${USER_INSTALL_ROOT}` as this path expands to the WebSphere Application Server path on your machine.
   e. Select a keystore type from the **Key Store Type** field.

      This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.
- Configure default key locators at the cell level.
  1. Connect to administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number.
  2. Click **Security > Web Services > Key Locators**.
  3. Click **New** to configure a new key locator; select the box next to a key locator name and click **Delete** to delete a key locator; or click the name of a

key locator to edit its configuration. If you are configuring a new key locator or editing an existing one, complete the following steps:

a. Specify a name for the key locator in the **Key Locator Name** field.
b. Specify a name for the key locator class implementation in the **Key Locator Classname** field.

   WebSphere Application Server has the following default key locator class implementations:

   com.ibm.wsspi.wssecurity.config.WSldKeyStoreMapKeyLocator
   This class, used by the response sender, maps an authenticated identity to a key. If encryption is used, this class is used to locate a key to encrypt the response message. The com.ibm.wsspi.wssecurity.config.WSldKeyStoreMapKeyLocator class has the capability to map an authenticated identity from the invocation credential of the current thread to a key that is used to encrypt the message. If an authenticated identity is present on the current thread, the class maps the ID to the mapped name. For example, user1 is mapped to mappedName_1. Otherwise, name="default". When a matching key is not found, the authenticated identity is mapped to the default key specified in the binding file.

   com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator
   This class, used by the response receiver, request sender, and request receiver, maps a name to an alias. Encryption uses this class to obtain a key to encrypt a message and digital signature uses this class to obtain a key to sign a message. The com.ibm.wsspi.wssecurity.config.KeyStoreKeyLocator class maps a logical name to a key alias in the key store file. For example, key #105115176771 is mapped to CN=Alice, O=IBM, c=US.

c. Specify the password used to access the keystore password in the **Key Store Password** field.

   This field is optional is the key locator does not use a keystore.

d. Specify the path name used to access the keystore in the **Key Store Path** field.

   This field is optional is the key locator does not use a keystore. Use ${*USER_INSTALL_ROOT*} as this path expands to the WebSphere Application Server path on your machine.

e. Select a keystore type from the **Key Store Type** field.

   This field is optional is the key locator does not use a keystore. Use the **JKS** option if you are not using Java Cryptography Extensions (JCE) and use **JCEKS** if you are using JCE.

Trusted ID evaluator

Trusted ID evaluator (com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator) is a abstraction of the mechanism that evaluates whether the given ID name is trusted. Depending upon the implementation, you can use various types of infrastructure to store a list of the trusted IDs, such as:

- Plain text file
- Database
- Lightweight Directory Access Protocol (LDAP) server

The trusted ID evaluator is typically used by the ultimate receiver in a multi-hop environment. The Web services security implementation invokes the trusted ID

evaluator and passes the identity name of the intermediary as a parameter. If the identity is evaluated and deemed trustworthy, the procedure continues. Otherwise, an exception is thrown and the procedure is aborted.

Trusted ID evaluator collection:

Use this page to view a list of trusted identity (ID) evaluators. The trusted ID evaluator determines whether to trust the identity-asserting authority. Once the ID is trusted, the WebSphere Application Server issues the proper credentials, which are used in a downstream call for invoking resources. The trusted ID evaluator implements the com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator interface.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Services, click **Web Services: Default bindings for Web Services Security > Trusted ID Evaluators**.

Click **New** to create a trusted ID evaluator.

Click **Delete** to a delete a trusted ID evaluator.

Using this panel, complete the following steps:
1. Specify a trusted ID evaluator name and trusted ID evaluator class name.
2. Save your changes by clicking **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.
3. Return to the Trusted ID Evaluator collection panel to update the Web services security run time with the default binding information, which is found in the `ws_security.xml` file.
4. Click **Update runtime**. The configuration changes made to the other Web services also are updated in the Web services security run time.

Trusted ID Evaluator Name:

Specifies the unique name of the trusted ID evaluator.

Trusted ID Evaluator Classname:

Specifies the class name of the trusted ID evaluator.

Trusted ID evaluator configuration settings:

Use this information to configure trust identity (ID) evaluators.

To view this administrative console page, complete the following steps:
1. Click **Servers > Application Servers >***server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Trust ID Evaluators > New**.

You must specify the name and value properties for the default trusted ID evaluator to create the trusted ID list for evaluation.

Trusted ID Evaluator Name:

Specifies the unique name used by the application binding to refer to a trusted identity (ID) evaluator defined in the default binding.

You must specify the trusted ID evaluator name in the form, `trustedId_n`, where`_n` is an integer from 0 to n.

Trusted ID Evaluator Class Name:

Specifies the class name of the trusted ID evaluator.

Default	com.ibm.wsspi.wssecurity. id.TrustedIDEvaluatorImpl

Login mappings

Login mappings, found in the `ibm-webservices-bnd.xmi` eXtended Markup Language (XML) file, contains a mapping configuration. This mapping configuration defines how the Web services security handler maps the token `<ValueType>` element, contained within the security token extracted from the message header, to the corresponding authentication method. The token `<ValueType>` element is contained within the security token extracted from a SOAP message header.
The sender-side Web services security handler generates and attaches security tokens based on `<AuthMethods>` element specified in the deployment descriptor. For example, if the authentication method is BasicAuth, the sender-side security handler generates and attaches `UsernameToken` (with both user name and password) to the SOAP message header. Web services security run time uses the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface as a security provider to generate security tokens on the client side (or when Web services is acting as client).

The sender security handler invokes the `handle()` method of a `javax.security.auth.callback.CallbackHandler` interface implementation. This implementation creates the security token and passes the token back to the sender security handler. The senders security handler constructs the security token based on the authentication information in the callback array. The security handler then inserts the security token into the Web Services Security message header.

The `CallbackHandlerinterface` implementation you use to generate the required security token is defined in the `<loginBinding>` element in the `ibm-webservicesclient-bnd.xmi` web services security binding file. For example,

```
<loginBinding xmi:id="LoginBinding_1052760331526" authMethod="BasicAuth"
      callbackHandler=
       "com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler"/>
```

The `<loginBinding>` element associates the `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler` interface with the BasicAuth authentication method. WebSphere Application Server provides the following set of CallbackHandler interface implementations you can use to create various security token types:

`com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
: If there is no basic authentication data defined in the login binding information (this is not the same as the HTTP basic authentication information), the previous token type prompts for user name and

password through a GUI login panel. It uses the basic authentication data defined in the login binding if it is defined. Use this CallbackHandler with the BasicAuth authentication method.

Attention: Do not use this CallbackHandler on the server because it prompts you for login binding information.

com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler
If basic authentication data is not defined in the login binding (this is not the same as the HTTP basic authentication information), it prompts for the user name and password using standard in (stdin). It uses the basic authentication data defined in the login binding if it is defined. Use this CallbackHandler with BasicAuth authentication method.

Attention: Do not use this CallbackHandler on the server because it prompts you for login binding information.

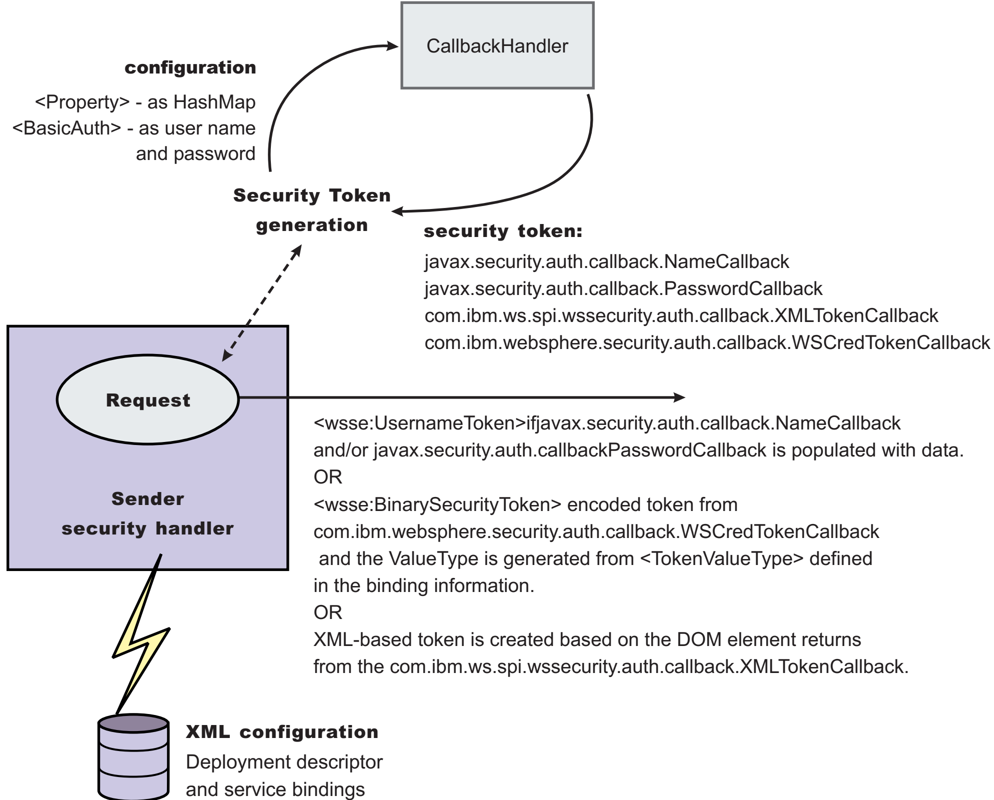
com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
This CallbackHandler does not prompt. Rather, it uses the basic authentication data defined in the login binding (this is not the same as the HTTP basic authentication information). This CallbackHandler is meant to be used with BasicAuth authentication method. This can be used when Web services is running as a client and needs to send basic authentication (`<wsse:UsernameToken>`) to the downstream call.

Attention: You must define the basic authentication data in the login binding information for this CallbackHandler.

com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler
The CallbackHandler generates Lightweight Third Party Authentication (LTPA) tokens from the run as `JAASSubject` (invocation Subject) of the current WebSphere Application Server security context. However, if there is basic authentication data defined in the login binding information (this is not the same as the HTTP basic authentication information), it uses the basic authentication data and LTPA token generated. The Web services security run time inserts the LTPA token as a binary security token (`<wsse:BinarySecurityToken>`) into the message SOAP header. The value type is mandatory. (See LTPA for more information). Use this CallbackHandler with the LTPA authentication method.

Attention: The **Token Type URI** and **Token Type Local Name** must be defined in the login binding information for this CallbackHandler. The token value type is used to validate the token to the request sender and request receiver binding configuration.

Figure.1 shows the sender security handler in the request sender message process.



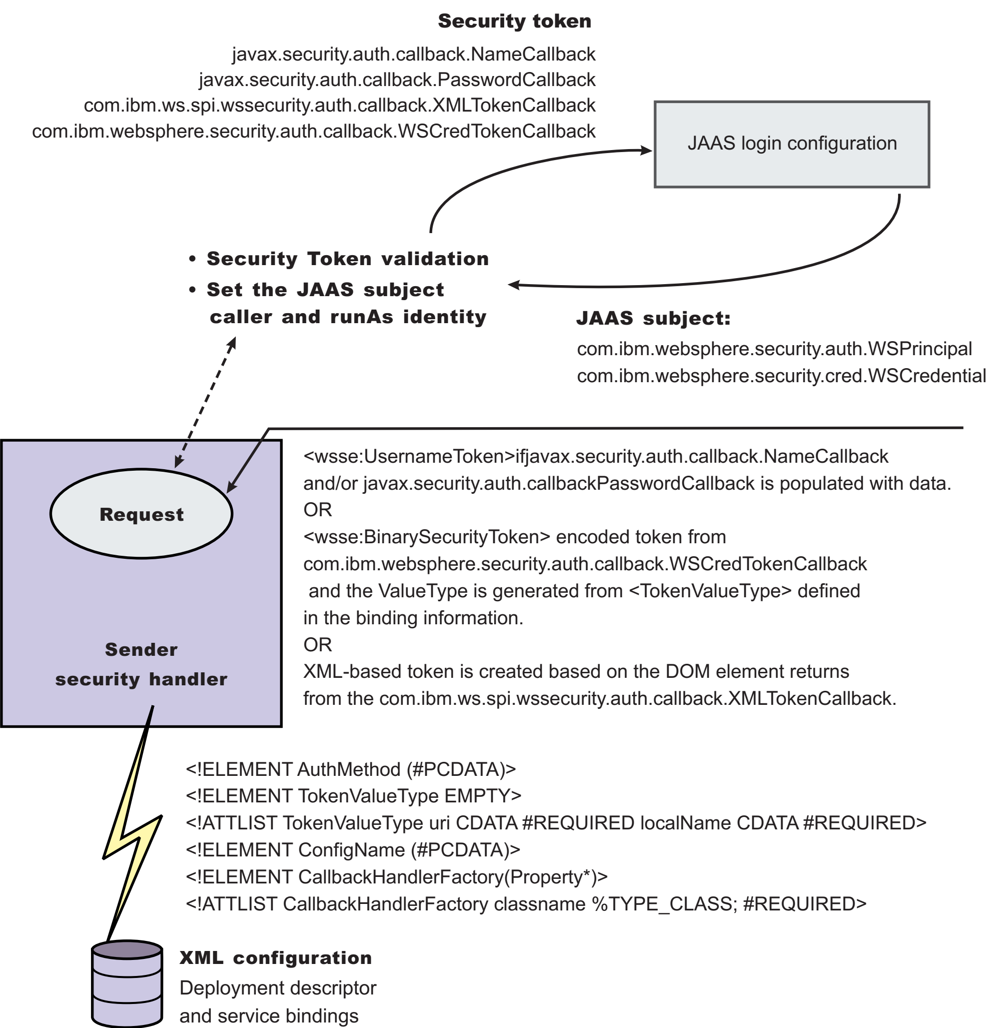
The receiver-side security server can be configured to support multiple authentication methods and multiple types of security tokens. Upon receiving a message, the receiver Web services security handler compares the token type (in the message header) with the expected token types configured in the deployment descriptor. The Web services security handler extracts the security token form the message header and maps the token `<ValueType>` element to the corresponding authentication method. The mapping configuration is defined in the `<loginMappings>` element in the `ibm-webservices-bnd.xmi` XML file. For example:

```
<loginMappings xmi:id="LoginMapping_1051977980074" authMethod="LTPA"
      configName="WSLogin">
     <callbackHandlerFactory xmi:id="CallbackHandlerFactory_1051977980081"
     classname="com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl"/>
      <tokenValueType xmi:id="TokenValueType_1051977980081"
      uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2" localName="LTPA"/>
</loginMappings>
```

The `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface is a factory for `javax.security.auth.callback.CallbackHandler`. The Web services security run time initiates the factory implementation class and passes the authentication information from Web services security header to the factory class through the set methods. The Web services security run time then invokes the `newCallbackHandler()` method to obtain the `javax.security.auth.CallbackHandler` object, which generates the required security token). When the security handler receives an LTPA `BinarySecurityToken`, it uses the `WSLogin` JAAS login configuration and the `newCallbackHandler()` method to validate the security token. If none of the expected token types are found in the SOAP message Web services security header, the request is rejected with an SOAP fault. Otherwise, the token type is used to map to a JAAS login configuration for token validation. If

authentication is successful, a JAAS Subject is created and associated with the thread of execution. Otherwise, the request is rejected with an SOAP fault.

Figure.2 shows the receiver security handler in the request receiver message process.



The following table shows the authentication methods and JAAS login configurations.

Authentication method	JAAS login configuration
BasicAuth	`WSLogin`
Signature	`system.wssecurity.Signature`
LTPA	`WSLogin`
IDAssertion	`system.wssecurity.IDAssertion`

The default `<LoginMapping>` is defined in the cell-level `ws-security.xml` and server-level `ws-security.xml` files. If nothing is defined in the binding file

information, the `ws-security.xml` default is used. However, an administrator can override the default by defining a new `<LoginMapping>` element in the binding file.

The client reads the default binding information in the `${WAS_HOME}/properties/Web Services Security.xml` file. The server run-time component loads the cell-level *Web Services Security.xml* and server-level *Web Services Security.xml* files if they exist.

The two files are merged in the run time to form one effective set of default binding information. On a base application server the server run time component only loads the server-level *Web Services Security.xml* file. The server-side *Web Services Security.xml* file and application Web services security binding information are managed by the administrative console and also by WSADMIN. You can specify the binding information during application deployment either through the administrative console or WSADMIN. The Web services security policy is defined in the deployment descriptor extension (`ibm-webservicesclient-ext.xmi`) and the bindings are stored in the IBM binding extension (`ibm-webservicesclient-bnd.xmi`). However, `${WAS_HOME}/properties/Web Services Security.xml` defines the default binding value for the client. If the binding information is not specified in the binding file, the run time reads the binding information from the default `${WAS_HOME}/properties/Web Services Security.xml` file.

Login mappings collection:

Use this page to view a list of configurations for validating security tokens within incoming messages. Login mappings map an authentication method to a Java Authentication and Authorization Service (JAAS) login configuration to validate the security token. Four authentication methods are predefined in the WebSphere Application Server: BasicAuth, Signature, IDAssertion, and LTPA.

To view this administrative console page, complete the following steps:

1. Click **Server > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings**.

Click **New** to create a login mapping.

Click **Delete** to delete a login mapping.

If you click **Update runtime**, the Web services security run time is updated with the default binding information, which is contained in the `ws-security.xml` file that was previously saved. After you specify the authentication method, the Java Authentication and Authorization Service (JAAS) configuration name, and the Callback Handler Factory class name on this panel, you must complete the following steps:

Click **Save** at the top of the administrative console. When you click **Save**, you return to the administrative console home panel.

Return to the Login Mappings collection panel and click **Update runtime**.

Important: When you click **Update runtime**, the configuration changes made to the other Web services also are updated in the Web services security run time.

Authentication Method:

Specifies the authentication method used for validating the security tokens.

The following authentication methods are available:

BasicAuth
: Basic authentication includes both a user name and password in the security token. The information in the token is authenticated by the receiving server and used to create a credential.

Signature
: When the authentication method is signature, an X.509 certificate is sent as a security token. For Lightweight Directory Access Protocol (LDAP) registries, the distinguished name (DN) is mapped to a credential, which is based on the LDAP certificate filter settings. For local OS registries, the first attribute of the certificate, usually the common name (CN) is mapped directly to a user ID in the registry.

IDAssertion
: Identity assertion maps a trusted identity (ID) to a WebSphere credential. This authentication method only includes a user name in the security token. An additional token is included in the message for trust purposes. Once the additional token is trusted, the IDAssertion token user name is mapped to a credential.

LTPA
: Lightweight Third Party Authentication (LTPA) validates an LTPA token.

JAAS Configuration Name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Callback Handler Factory Class Name:

Specifies the name of the factory for the CallbackHandler class.

Login mapping configuration settings:

Use this page to specify the Java Authentication and Authorization Service (JAAS) login configuration settings used to validate security tokens within incoming messages.

To view this administrative console page, complete the following steps:

1. Click **Servers > Application Servers >** *server_name*.
2. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Login Mappings > New**.

Authentication Method:

Specifies the method of authentication.

You can use any string, but the string must match the element in the service-level configuration. The following words are reserved and have special meanings:

BasicAuth
: Uses both a user name and a password

IDAssertion
: Uses only a user name, but requires that additional trust is established on the receiving server using a TrustedIDEvaluator

Signature
Uses the distinguished name (DN) of the signer.

LTPA Validates a token

JAAS Configuration Name:

Specifies the name of the Java Authentication and Authorization Service (JAAS) configuration.

Specify your JAAS configurations using the administrative console by clicking **Security > JAAS Configuration > Application**.

Callback Handler Factory Class Name:

Specifies the name of the factory for the CallbackHandler class.

You must implement the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` class in this field.

Default:	com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory

Token Type URI:

Specifies the namespace Uniform Resource Identifiers (URI), which denotes the type of security token accepted.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType element identifies the type of security token and its namespace. If eXtensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the **Authentication Method** field, this field is ignored.

Data type:	Unicode characters except for non-ASCII characters, but including the number sign (#), the percent sign (%), and the square brackets ([ ]).

Token Type Local Name:

Specifies the local name of the security token type, for example, X509v3.

If binary security tokens are accepted, the value denotes the ValueType attribute in the element. The ValueType attribute identifies the type of security token and its namespace. If eXtensible Markup Language (XML) tokens are accepted, the value denotes the top-level element name of the XML token.

If the reserved words are specified previously in the **Authentication Method** field, this field is ignored.

Nonce Maximum Age:

Specifies the time, in seconds, before the nonce time stamp expires. Nonce is a randomly generated value.

You must specify a minimum of 300 seconds for the **Nonce Maximum Age** field. However, the maximum value cannot exceed the number of seconds specified in the **Nonce Cache Timeout** field for either the server level or the cell level. You can specify the **Nonce Maximum Age** value for the server level by clicking **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**. You can specify the **Nonce Maximum Age** value for the cell level by clicking **Security > Web Services > Properties**.

Important: The **Nonce Maximum Age** field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth.` If you specify BasicAuth, but do not specify values for the **Nonce Maximum Age** field, the Web services security run time searches for a **Nonce Maximum Age** value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 300 seconds.

Default	300 seconds
Range	300 to **Nonce Cache Timeout** seconds

Nonce Clock Skew:

Specifies the clock skew value, in seconds, to consider when WebSphere Application Server checks the freshness of the message. Nonce is a randomly generated value.

You must specify a minimum of 0 seconds for the **Nonce Clock Skew** field. However, the maximum value cannot exceed the number of seconds specified in the **Nonce Maximum Age** on this Login Mappings panel.

Important: The **Nonce Clock Skew** field on this panel is optional and only valid if the BasicAuth authentication method is specified. If you specify another authentication method and attempt to specify values for this field, the following error message displays and you must remove the specified value: `Nonce is not supported for authentication methods other than BasicAuth.` If you specify BasicAuth, but do not specify values for the **Nonce Clock Skew** field, the Web services security run time searches for a **Nonce Clock Skew** value on the server level. If a value is not found on the server level, the run time searches the cell level. If a value is not found on either the server level or the cell level, the default is 0 seconds.

Default	0 seconds
Range	0 to **Nonce Maximum Age** seconds

Configuring the client for request signing: digitally signing message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the WebSphere Application Server Toolkit.

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the WebSphere Application Server Toolkit.
6. Expand **Request Sender Configuration > Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.
7. Select the parts of the message in which to sign by clicking **Add** and selecting one of the following three parts: body, timestamp, or SecurityToken The following is a list and description of the message parts

   Body This is the user data portion of the message.

   Timestamp
   The timestamp determines if the message is valid based on the time the message was sent and then received. If **timestamp** is selected, proceed to the next step to **Add Created Time Stamp** to the message.

   Securitytoken
   The security token authenticates the client. If **securitytoken** is selected, the message is signed.

   You can choose to digitally sign the message using a time stamp if **Add Created Time Stamp** is selected and configured. You can digitally sign the message using a security token if a login configuration authentication method is selected.
8. Optional: Expand the **Add Created Time Stamp** section and select this option if you want a time stamp added to the message You can specify an expiration time for the time stamp, which helps defend against replay attacks.

   The lexical representation for duration is the [ISO 8601] extended format *PnYnMnDTnHnMnS*, where:

   - *nY* represents the number of years

- *nM* represents the number of months
- *nD* represents the number of days
- *T* is the date and time separator
- *nH* represents the number of hours
- *nM* represents the number of minutes
- *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

For example, to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, the format is: `P1Y2M3DT10H30M`. Typically, configure a message time stamp for about 10 to 30 minutes, which is `P0Y0M0DT0H10M0S`.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor**field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the client sends a message to a server.

Once you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See Configuring the client for request signing: choosing the digital signature method for more information.

Configuring the client for request signing: choosing the digital signature method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively. Also, you must specify which parts of the message sent by the client must be digitally signed. See Configuring the client for request signing: digitally signing message parts for more information.

Complete the following steps to specify which message parts to digitally sign when configuring the client for request signing:

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab.
6. Expand **Security Request Sender Binding Configuration > Signing Information**.
7. Select **Edit** to view the signing information and select a digital signature method from the **Signature method algorithm** field. The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. The signing of the `<DigestValue>` element binds resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

Name	Purpose
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation.

The signing key name refers to a key entry associated with the signing key locator. The key entry has an alias, which is found in the key store or wherever the certificates are stored based upon the key locator implementation. The signing key locator references the implementation class that locates the correct keystore where the alias and certificate exists.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor**field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the

request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method is used to digitally sign a message when the client sends a message to a server.

Once you have configured the client to digitally sign the message, you must configure the server to verify the digital signature. See Configuring the server for request digital signature verification: verifying the message parts for more information.

Configuring the server for request digital signature verification: verifying the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which parts of the message sent by the client must be digitally signed. See Configuring the client for request signing: digitally signing message parts to determine which message parts are digitally signed. The message parts specified for the client request sender must match the message parts specified for the server request receiver.

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which parts of the request to verify.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Security Extensions** tab in the Web Services Editor.
6. Expand the **Request Receiver Service Configuration Details > Required Integrity** section. Required integrity refers to the parts of the message that require digital signature verification. The purpose of digital signature verification is to make sure that the message parts have not been modified while it was transmitted across the Internet.
7. Select the parts of the message to verify by clicking **Add** and selecting one of the following three parts: body, timestamp, or SecurityToken. You can determine which parts of the message to verify by looking at the Web Service Request Sender Configuration in the client application. To view the Web Service Request Sender Configuration information in the Web Services Client Editor, click the **Security Extensions** tab and expand **Request Sender Configuration > Integrity**. The following information is a list and description of the message parts:

   Body This is the user data portion of the message.

Timestamp
: The timestamp determines if the message is valid based on the time the message is sent and then received. If **timestamp** is selected, proceed to the next step to **Add Created Time Stamp** to the message.

Securitytoken
: The security token authenticates the client. If **securitytoken** is selected, the message is signed.

8. Optional: Expand the **Add Received Time Stamp** section. The Add Received Time Stamp indicates to validate the Add Created Time Stamp configured by the client. You must select option this if you selected Add Created Time Stamp on the client. The time stamp ensures message integrity by indicating the freshness of the request. This option helps to defend against replay attacks.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the server when the client sends a message to a server.

After you have specified what message parts contain a digital signature that must be verified by the server, you must configure the server to recognize the digital

signature method used to digitally sign the message. See Configuring the server for request digital signature verification: choosing the verification method for more information.

Configuring the server for request digital signature verification: choosing the verification method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the server. See Configuring the server for request digital signature verification: verifying the message parts to specify which message parts are digitally signed by the client and must be verified by the server. The message parts specified for the client request sender must match the message parts specified for the server request receiver. Likewise, the digital signature method chosen for the client must match the digital signature method used by the server.

Complete the following steps to configure the server for request digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the server will use during verification.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab.
6. Expand the **Security Request Receiver Binding Configuration Details > Signing Information** section.
7. Click **Edit** to edit the signing information. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Use certificate path reference**
   - **Trust anchor reference**
   - **Certificate store reference**
   - **Trust any certificate**

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the server request receiver configuration must match the algorithm selected in the client request sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is sent with the message. This public key or certificate might not be validated at the receiving end. By selecting **User certificate path reference**, you must configure a trust anchor reference and a certificate store reference to validate the certificate sent with the message. By selecting **Trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a key store that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application. See

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the server uses to verify the digital signature in the message parts.

After you configure the client for request signing and the server for request digital signature verification, you must configure the server and the client to handle the response. Next, specify the response signing for the server. See Configuring the server for response signing: digitally signing message parts for more information.

Configuring the server for response signing: digitally signing message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which message parts to digitally sign when configuring the server for response signing:

1. Launch the WebSphere Application Server Toolkit and either click**Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.

2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the WebSphere Application Server Toolkit.
6. Expand **Response Sender Service Configuration Details > Integrity**. Integrity refers to digital signature while confidentiality refers to encryption. Integrity decreases the risk of data modification while the data is transmitted across the Internet. For more information on digitally signing SOAP messages, see XML digital signature.
7. Select the parts of the message in which to sign by clicking **Add** and selecting **Body**, **Timestamp**, or **Securitytoken**.

   Body The body is the user data portion of the message.

   Timestamp
   The timestamp determines if the message is valid based on the time that the message is sent and then received. If **timestamp** if selected, proceed to the next step to **Add Created Time Stamp** to the message.

   Securitytoken
   If security token is selected, the authentication information is added to the message.
8. Optional: Expand the **Add Created Time Stamp** section. Select this option if you want to add a time stamp to the message. Also, you can specify an expiration time for the time stamp, which helps defend against replay attacks. The lexical representation for duration is the ISO 8601 extended format, *PnYnMnDTnHnMnS*, where:
   - *nY* represents the number of years.
   - *nM* represents the number of months.
   - *nD* represents the number of days.
   - *T* is the date and time separator.
   - *nH* represents the number of hours.
   - *nM* represents the number of minutes.
   - *nS* represents the number of seconds. The number of seconds can include decimal digits to arbitrary precision.

   For example, if you want to indicate a duration of 1 year, 2 months, 3 days, 10 hours, and 30 minutes, write: `P1Y2M3DT10H30M`. Typically, a message time stamp is configured for about 10 to 30 minutes. 10 minutes is represented as: `P0Y0M0DT0H10M0S`.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:
- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts to digitally sign when the server sends a response to the client.

Once you have specified which message parts to digitally sign, you must specify which method is used to digitally sign the message. See Configuring the server for response signing: choosing the digital signature method for more information.

Configuring the server for response signing: choosing the digital signature method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the WebSphere Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which digital signature method to use when configuring the server for response signing:

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
5. Click the **Binding Configurations** tab.

6. Expand **Response Sender Binding Configuration Details > Signing Information**.
7. Click **Edit** to choose a signing method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   The following table describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation. The same algorithm used here should also be used on the client response receiver. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>` element. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.
Signing key name	The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.
Signing key locator	The signing key locator represents a reference to a key locator implementation. For more information on configuring key locators, see any of the following files: • Configuring key locators using the WebSphere Application Server Toolkit • Configuring key locators using the administrative console • Configuring server and cell level key locators using the administrative console

The **Signing key name** refers to a key entry associated with the signing key locator. The key entry has an alias, which is found in the keystore or wherever the certificates are stored based upon the key locator implementation. The **Signing key locator** references the implementation class that locates the correct key store where the alias and certificate exists.

You have specified which method is used to digitally sign a message when the server sends a message to a client.

Once you have configured the server to digitally sign the response message, you must configure the client to verify the digital signature contained in the response message. See Configuring the client for response digital signature verification: verifying the message parts for more information.

Configuring the client for response digital signature verification: verifying the message parts

Prior to completing these steps, read either of the following topics to becomes familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Servers Client Editor within the WebSphere Application Server Toolkit:

- Configuring the client security bindings using the WebSphere Application Server Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which parts of the response to verify.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Security Extensions** tab.
6. Expand the **Response Receiver Configuration > Required Integrity** section. Required Integrity refers to parts that require digital signature verification. Digital signature verification decreases the risk that the message parts have been modified while the message is transmitted across the Internet.
7. Select the parts of the message that must be verified. You can determine which parts of the message to select by looking at the Web service response sender configuration. To add parts of the message, click **Add** and select one of the following three parts:

   Body This is the user data portion of the message.

   Timestamp
   The time stamp determines if the message is valid based on the time the message was sent and then received. If timestamp is selected, proceed to the next step to **Add Received Time Stamp** to the message.

Securitytoken
: The security token authenticates the client. If **Securitytoken** is selected, the message is signed.

8. Optional: Expand the **Add Received Time Stamp** section. Select **Add Received Time Stamp** to add the received time stamp to the message.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back. Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which message parts are digitally signed and must be verified by the client when the server sends a response message to the client.

After you have specified which message parts contain a digital signature that must be verified by the client, you must configure the client to recognize the digital signature method used to digitally sign the message. See Configuring the client for response digital signature verification: choosing the verification method for more information.

Configuring the client for response digital signature verification: choosing the verification method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Editor within the Application Server Toolkit:

- Configuring the server security bindings using the WebSphere Application Server Toolkit
- Configuring the server security bindings using the administrative console

You can use these two tabs to configure the Web Services Security extensions and Web Services Security bindings, respectively. Also, you must specify which message parts contain digital signature information that must be verified by the client. See Configuring the client for response digital signature verification: verifying the message parts to specify which message parts are digitally signed by the server and must be verified by the client. The message parts specified for the server response sender must match the message parts specified for the client response receiver. Likewise, the digital signature method chosen for the server must match the digital signature method used by the client.

Complete the following steps to configure the client for response digital signature verification. The steps describe how to modify the extensions to indicate which digital signature method the client will use during verification.

1. Launch the WebSphere Application Server Toolkit and either click **Windows > Open Prospective > Java** or **Windows > Open Prospective > Resource**.
2. Select the Web services enabled EJB or Web module.
3. In the Package Explorer window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
4. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
5. Click the **Port Binding** tab.
6. Expand the **Security Response Receiver Binding Configuration > Signing Information** section.
7. Click **Edit** to select a digital signature method. The signing info dialog displays and either select or enter the following information:
   - **Canonicalization method algorithm**
   - **Digest method algorithm**
   - **Signature method algorithm**
   - **Signing key name**
   - **Signing key locator**

   For more conceptual information on digitally signing SOAP messages, see XML digital signature. The following table describes the purpose for each of these selections. Some of the following definitions are based on the XML-Signature specification, which can be found at: http://www.w3.org/TR/xmldsig-core.

Name	Purpose
Canonicalization method algorithm	The canonicalization method algorithm is used to canonicalize the `<SignedInfo>` element before it is digested as part of the signature operation.

Name	Purpose
Digest method algorithm	The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the `<DigestValue>` binds resource content to the signer key. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Signature method algorithm	The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` element into the `<SignatureValue>` element. The algorithm selected for the client response receiver configuration must match the algorithm selected in the server response sender configuration.
Use certificate path reference or **Trust any certificate**	When a message is signed, the public key used to sign it is transmitted with the message. To validate this public key at the receiving end, configure a certificate path reference. By selecting **User certificate path reference,** you must configure a trust anchor reference and certificate store reference to validate the certificate sent with the message. By selecting **trust any certificate**, the signature is validated by the certificate sent with the message without the certificate itself being validated.
Use certificate path reference: Trust anchor reference	A trust anchor is a configuration that refers to a keystore that contains trusted, self-signed certificates and certificate authority (CA) certificates. These certificates are trusted certificates that you can use with any applications in your deployment.
Use certificate path reference: Certificate store reference	A certificate store is a configuration that has a collection of X.509 certificates. These certificates are not trusted for all applications in your deployment, but might be used as an intermediary to validate certificates for an application.

Important: If you configure the client and server signing information correctly, but receive a `Soap body not signed` error when executing the client, you might need to configure the actor. You can configure the actor in the following locations on the client in the Web Services Client Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Client Service Configuration Details** and indicate the actor information in the **Actor URI** field.
- Click **Security Extensions > Request Sender Configuration > Details** and indicate the actor information in the **Actor** field.

You must configure the same actor strings for the Web service on the server, which processes the request and sends the response back.

Configure the actor in the following locations in the Web Services Editor within the WebSphere Application Server Toolkit:

- Click **Security Extensions > Server Service Configuration**.
- Click **Security Extensions > Response Sender Service Configuration Details > Details** and indicate the actor information in the **Actor** field.

The actor information on both the client and server must refer to the same exact string. When the actor fields on the client and server match, the request or response is acted upon instead of being forwarded downstream. The **actor** fields might be different when you have Web services acting as a gateway to other Web services. However, in all other cases, make sure that the actor information matches on the client and server. When Web services are acting as a gateway and they do not have the same actor configured as the request passing through the gateway, Web services do not process the message from a client. Instead, these Web services send the request downstream. The downstream process that contains the correct actor string processes the request. The same situation occurs for the response. Therefore, it is important that you verify that the appropriate client and server actor fields are synchronized.

You have specified which method the client uses to verify the digital signature in the message parts.

After you configure the server for response signing and the client for request digital signature verification, verify that you have configured the client and the server to handle the message request.

Configuring the client security bindings using the Assembly Toolkit

When configuring a client for Web services security, the bindings describe how to execute the security specifications found in the extensions. Use the Web Services Client Editor within the Assembly Toolkit to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a Web service or from a Web service accessing a downstream Web service. This document focuses on the pure client situation. However, the concepts, and in most cases the steps, also apply when a Web service is configured to communicate downstream to another Web service that has client bindings. Complete the following steps to edit the security bindings on a pure client (or server acting as a client) using the Assembly Toolkit:

1. Import the Web services client EAR file into the Assembly Toolkit. When you edit the client bindings on a server acting as a client, the same basic steps apply. Complete the following steps to import your client EAR file into the Assembly Toolkit. Refer to the Assembly Toolkit documentation for additional information.
   a. Download and install the Assembly Toolkit. You can download the Assembly Toolkit from the following Web site: http://www.ibm.com /support/docview.wss?rs=180&context=SSEQTP&q= ASTK&uid=swg24005125&loc= en_US&cs =utf-8&lang=en+en
   b. Start the Assembly Toolkit and open the Java Perspective by selecting **Window > Open Perspective > J2EE**.

c. Import the client EAR file by selecting **File > Import > EAR file**.
d. Click **Next**.
e. Enter the path name to the EAR file in the **EAR File** field or click **Browse** to locate the file.
f. Enter the project name in the **Project name** field.
g. Click **Finish**.

2. Open the Web Services Client Editor within the Assembly Toolkit to begin editing the client bindings. To access the client bindings using the Assembly Toolkit, complete the following steps:
   a. Open the Navigator by clicking **Window > Show View > Navigator**.
   b. Expand your application Java archive (JAR) from the Navigator.
   c. Expand the J2EE client application (appClientModule, ejbModule, or WebContent), which should be part of the client JAR package that you selected.
   d. Expand the **META-INF** directory and locate the `webservicesclient.xml` file.
   e. Right-click the `webservicesclient.xml` file and click **Open With > Web Services Client Editor**. In the Web Services Client Editor (for `webservicesclient.xml` and outbound requests and inbound responses Web services configuration), there are several tabs at the bottom of the editor including **Service References**, **Handlers**, **Security Extensions**, **Web Services Client Binding**, and **Port Binding**. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the **Port Binding** tab.

3. On the **Security Extensions** tab, select the port qualified name bindings that you want to configure. The Web services security extensions are configured for outbound requests and inbound responses. You need to configure the following information for Web services security extensions. These topics are discussed in more detail in other sections of the documentation.

   Request sender configuration details

   Details
   : Configuring the client for request signing: digitally signing message parts

   Integrity
   : Configuring the client for request signing: digitally signing message parts

   Confidentiality
   : Configuring the client for request encryption: encrypting the message parts

   Login Config

   BasicAuth
   : Configuring the client for basicauth authentication: specifying the method

   IDAssertion
   : Configuring the client for identity assertion authentication: specifying the method

   Signature
   : Configuring the client for signature authentication: specifying the method

LTPA Configuring the client for LTPA token authentication: specifying LTPA token authentication

ID Assertion
: Configuring the client for identity assertion authentication: specifying the method

Add Created Time Stamp
: Configuring the client for request signing: digitally signing message parts

Response receiver configuration details

Required Integrity
: Configuring the client for response digital signature verification: verifying the message parts

Required Confidentiality
: Configuring the client for response decryption: decrypting message parts

Add Received Time Stamp
: Configuring the client for response digital signature verification: verifying the message parts

4. From the **Port Binding** tab, select the port qualified name binding that you want to configure. The Web services security bindings are configured for outbound requests and inbound responses. You need to configure the following information for Web services security bindings. These topics are discussed in more details in other sections of the documentation.

Security request sender binding configuration

Signing information
: Configuring the client for request signing: choosing the digital signature method

Encryption information
: Configuring the client for request encryption: choosing the encryption method

Key locators
: Configuring key locators using the Assembly Toolkit

Login binding

Basic auth
: Configuring the client for basicauth authentication: collecting the authentication information

ID assertion
: Configuring the client for identity assertion: Collecting the authentication method

Signature
: Configuring the client for signature authentication: collecting the authentication information

LTPA Configuring the client for LTPA token authentication: Collecting the authentication method information

Security response receiver binding configuration

Signing information
: Configuring the client for response digital signature verification: choosing the verification method

Encryption information
: Configuring the client for response decryption: choosing a decryption method

Trust anchor
: Configuring trust anchors using the Assembly Toolkit

Certificate store list
: Configuring the client-side collection certificate store using the Application Server Toolkit

Key locators
: Configuring key locators using the Assembly Toolkit

Important: When configuring the Security Request Sender Binding Configuration, you must synchronize the information used to perform the specified security with the Security Request Receiver Binding Configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects as there is no negotiation during run time to figure out the requirements of the server. For example, when configuring the encryption information in the Security Request Sender Binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This is an example of how the client and server configuration are tightly coupled. Additionally, when configuring the Security Response Receiver Binding Configuration, the server must send the response using security information known by this client Security Response Receiver Binding Configuration.

The following table shows the related configurations between the client and server. The client request sender and server request receiver are relative configurations that must be synchronized with each other. The server response sender and client response receiver are related configurations that must be synchronized with each other. Note that related configurations are end points for any request or response. One end point must communicate its actions with the other end point because they do not negotiate any run time requirements.

Table 3. Related configurations

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the security bindings on a server acting as a client using the administrative console

When configuring a client for Web services security, the bindings describe how to execute the security specifications found in the extensions. Use the Web Services Client Editor within the WebSphere Application Server Toolkit to include the binding information in the client enterprise archive (EAR) file.

You can configure the client-side bindings from a pure client accessing a Web service or from a Web service accessing a downstream Web service. Complete the following steps to find the location in which to edit the client bindings from a Web service that is running on the server. When a Web service communicates with another Web service, you must configure client bindings to access the downstream Web service.

1. Deploy the Web service using the WebSphere Application Server administrative console by clicking **Applications > Install New Application**. You can access the administrative console by typing `http://localhost:9090/admin` in your Web browser unless you have changed the port number. For more information on installing an application, see Installing a new application
2. Click **Applications > Enterprise Applications >** *application_name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending upon which type of service is the client to the downstream service.
   - For Web Modules, click the Web Archive (WAR) file that you configured as the client.
   - For EJB Modules, click the Java Archive (JAR) file that you configured as the client.
4. Click the name of the WAR or JAR file.
5. Under Additional Properties, click **Web Services: Client Security Bindings**. A table displays with the following columns:
   - **Component Name**
   - **Port**
   - **Web Service**
   - **Request Sender Binding**
   - **Request Receiver Binding**
   - **HTTP Basic Authentication**
   - **HTTP SSL Configuration**

   For Web services security, you must edit the Request Sender Binding and Response Receiver Binding configurations. You can use the defaults for some of the information at the server level (and at the cell level for some information). Default bindings are convenient because you can configure commonly reused elements such as key locators once and then reference their aliases in the application bindings.
6. View the default bindings for the server using the Administrative Console by clicking **Servers > Application Server >** *server1*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security**. You can configure the following sections. These topics are discussed in more detail in other sections of the documentation.
   - Request sender binding
     - Signing information
     - Encryption information
     - Key locators
     - Login bindings
   - Response receiver binding
     - Signing information
     - Encryption information
     - Trust anchors
     - Collection certificate store

– Key locators

Important: When configuring the Security Request Sender Binding Configuration, you must synchronize the information used to perform the specified security with the Security Request Receiver Binding Configuration, which is configured in the server EAR file. These two configurations must be synchronized in all respects as there is no negotiation during run time to figure out the requirements of the server. For example, when configuring the encryption information in the Security Request Sender Binding Configuration, you must use the public key from the server for encryption. Therefore, the key locator that you choose must contain the public key from the server configuration. The server must contain the private key to decrypt the message. This is an example of how the client and server configuration are tightly coupled. Additionally, when configuring the Security Response Receiver Binding Configuration, the server must send the response using security information known by this client Security Response Receiver Binding Configuration.

The following table shows the related configurations between the client and server. The client request sender and server request receiver are relative configurations that must be synchronized with each other. The server response sender and client response receiver are related configurations that must be synchronized with each other. Note that related configurations are end points for any request or response. One end point must communicate its actions with the other end point because they do not negotiate any run time requirements.

Table 4. Related configurations

Client configuration	Server configuration
Request sender	Request receiver
Response receiver	Response sender

Configuring the server security bindings using the Assembly Toolkit

Create an enterprise JavaBean (EJB) file Java archive (JAR) file or Web archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the WSDL2Java command. You can edit these files using the Web Services Editor in the Assembly Toolkit.

When configuring server-side security for Web services security, the security extensions configuration specifies what security is to be performed while the security bindings configuration indicates how to perform what is specified in the security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference them from the WAR and JAR binding configurations.

Prior to importing the Web services enterprise archive (EAR) file into the Assembly Toolkit, make sure that you have already run the `wsdl2java` command on your Web service to enable your J2EE application. You must import the Web services

enterprise archive (EAR) file into the Assembly Toolkit. Complete the following steps to import your EAR file into the Assembly Toolkit:

1. Download, install, and launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Import the application EAR file by clicking **File > Import > EAR file**.
4. Click **Next** and indicate both the EAR file name in the **EAR File** field and the project name in the **Project name** field.
5. Click **Finish**.

Refer to Assembly Toolkit documentation for more information.

Open the Web Services Editor in the Assembly Toolkit to begin editing the server security extensions and bindings. The following steps will help you locate the server security extensions and bindings. Other tasks specify how to configure each section of the extensions and bindings in more detail.

1. Expand your application module from the Navigator. If the Navigator is not shown, you can open it by clicking **Window > Show View > Navigator**.
2. If your application is a Web application (WAR) file, the following steps apply:
   a. Expand the **WebContent > WEB-INF** section.
   b. Locate the **webservices.xml** file. The **webservices.xml** file represents the server-side (inbound) Web services configuration. The **webservicesclient.xml** file represents the client-side (outbound) Web services configuration.
      1) To configure the server for inbound requests and outbound responses security configuration, right-click the **webservices.xml** file and click **Open With > Web Services Editor**.
      2) To configure the client for outbound requests and inbound responses security configuration, right-click the **webservicesclient.xml** file and click **Open With > Web Services Client Editor**. For more information, see Configuring the client security bindings using the Assembly Toolkit.
3. If your application is an EJB Application (JAR), the following steps apply:
   a. Expand the **ejbModule > META-INF** section.
   b. Locate the **webservices.xml** file. The **webservices.xml** file represents the server-side (inbound) Web services configuration. The **webservicesclient.xml** file represents the client-side (outbound) Web services configuration.
      1) To configure the server for inbound requests and outbound responses security configuration, right-click the **webservices.xml** file and click **Open With > Web Services Editor**.
      2) To configure the client for outbound requests and inbound responses security configuration, right-click the **webservicesclient.xml** file and click **Open With > Web Services Client Editor**. For more information, see Configuring the client security bindings using the Assembly Toolkit.
4. In the Web Services Editor (for **webservices.xml** and inbound requests and outbound responses Web services configuration), there are several tabs at the bottom of the editor including **Web Services**, **Port Components**, **Handlers**, **Security Extensions**, **Bindings**, and **Binding Configurations**. The security extensions are edited using the **Security Extensions** tab. The security bindings are edited using the **Security Bindings** tab.

a. On the **Security Extensions** tab, select the port component binding that you want to edit. The Web services security extensions are configured for inbound requests and outbound responses. You need to configure the following information for Web services security extensions. These topics are discussed in more detail in other sections of the documentation.

Request receiver service configuration details

Required integrity
: Configuring the server for request digital signature verification: verifying the message parts

Required confidentiality
: Configuring the server for request decryption: decrypting message parts

Login config

- **Basic auth**
  : Configuring the server to handle basicauth authentication
- **ID assertion**
  : Configuring the server to handle identity assertion authentication
- **Signature**
  : Configuring the server to handle signature authentication
- **LTPA** Configuring the server to handle LTPA token authentication

Add received time stamp
: Configuring the server for request digital signature verification: verifying the message parts

Response sender service configuration details

Details
: Configuring the server for response signing: digitally signing message parts

Integrity
: Configuring the server for response signing: digitally signing message parts

Confidentiality
: Configuring the server for response encryption: encrypting message parts

Add created time stamp-
: Configuring the server for response signing: digitally signing message parts

b. On the **Binding Configurations** tab, select the port component binding that you want to edit. The Web services security bindings are configured for inbound requests and outbound responses. You need to configure the following information for Web services security bindings. These topics are discussed in more details in other sections of the documentation.

Response receiver binding configuration details

Signing Information
: Configuring the server for request digital signature verification: choosing the verification method

Encryption Information
: Configuring the server for request decryption: choosing the decryption method

Trust Anchor
: Configuring trust anchors using the Assembly Toolkit

Certificate Store List
: Configuring the server-side collection certificate store using the Assembly Toolkit

Key Locators
: Configuring key locators using the Assembly Toolkit

Login Mapping

Basic auth
: Configuring the server to validate basicauth authentication information

ID assertion
: Configuring the server to validate identity assertion authentication information

Signature
: Configuring the server to validate signature authentication information

LTPA
: Configuring the server to validate LTPA token authentication information

Trusted ID Evaluator

Trusted ID Evaluator Reference

Response sender binding configuration details

Signing information
: Configuring the server for response signing: choosing the digital signature method

Encryption information
: Configuring the server for response encryption: choosing the encryption method

Key Locators
: Configuring key locators using the Assembly Toolkit

Configuring the server security bindings using the Administrative Console

Create an enterprise JavaBean (EJB) file Java archive (JAR) file or Web archive (WAR) file containing the security binding file (`ibm-webservices-bnd.xmi`) and the security extension file (`ibm-webservices-ext.xmi`). If this archive is acting as a client to a downstream service, you also need the client-side binding file (`ibm-webservicesclient-bnd.xmi`) and the client-side extension file (`ibm-webservicesclient-ext.xmi`). These files are generated using the WSDL2Java command. You can edit these files using the Web Services Editor in the Assembly Toolkit.

When configuring server-side security for Web services security, the security extensions configuration specifies what security is to be performed while the security bindings configuration indicates how to perform what is specified in the

security extensions configuration. You can use the defaults for some elements at the cell and server levels in the bindings configuration, including key locators, trust anchors, the collection certificate store, trusted ID evaluators, and login mappings and reference them from the WAR and JAR binding configurations.

The following steps describe how to edit bindings for a Web service after they are deployed on a server. When one Web service communicates with another Web service, you also must configure the client bindings to access the downstream Web service.

1. Deploy the Web service using the WebSphere Application Server Administrative Console. The Administrative Console is accessible by typing `http://localhost:9090/admin` in a Web browser. Once you have logged into the Administration Console, click **Applications > Install New Application** to deploy the Web service. For more information, see Installing a new application.
2. Once you have deployed the Web service, click **Applications > Enterprise Applications >** *application _name*.
3. Under Related Items, click either **Web Modules** or **EJB Modules** depending on which service you want to configure.
   a. If you select **Web Modules**, click the WAR file that you want to edit.
   b. If you select **EJB Modules**, click the JAR file that you want to edit.
4. Once you select a WAR or JAR file, under **Additional Properties**, click **Web Services: Client Security Bindings** for outbound requests and inbound responses or click **Web Services: Server Security Bindings** for inbound requests and outbound responses.
5. If you click **Web Services: Server Security Bindings**, the following sections can be configured. These topics are discussed in more details in other sections of the documentation.
   - Request receiver binding
     - Signing Information
     - Encryption Information
     - Trust anchors
     - Collection certificate store
     - Key locators
     - Trusted ID evaluators
     - Login mappings
   - Response sender binding
     - Signing Information
     - Encryption Information
     - Key locators

XML encryption

XML Encryption is a specification developed by W3C in 2002 that contains the steps to encrypt data; the steps to decrypt encrypted data; the XML syntax to represent encrypted data; the information used to decrypt the data; and a list of encryption algorithms such as triple DES, AES, and RSA.

You can apply XML encryption to an XML element, XML element content, and arbitrary data, including an XML document. For example, suppose that you need to encrypt the `CreditCard` element shown in the example 1.

Example 1: Sample XML document

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Example 2: XML document with a common secret key

Example 2 shows the XML document after encryption. The EncryptedData element represents the encrypted CreditCard element. The EncryptionMethod element describes the applied encryption algorithm, which is triple DES in this example. The KeyInfo element contains the information to retrieve a decryption key, which is a KeyName element in this example. The CipherValue element contains the ciphertext obtained by serializing and encrypting the CreditCard element.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
       Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
       <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
         <KeyName>John Smith</KeyName>
       </KeyInfo>
       <CipherData>
         <CipherValue>ydUNqHkMrD...</CipherValue>
       </CipherData>
  </EncryptedData>
</PaymentInfo>
```

Example 3: XML document encrypted with the public key of the recipient

In example 2, it is assumed that both the sender and recipient have a common secret key. If the recipient has a public and private key pair, which is a most likely the case, the CreditCard element can be encrypted as shown in example 3. The EncryptedData element is the same as the EncryptedData element found in example 2. However, the KeyInfo element contains an EncryptedKey.

```
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <EncryptionMethod
      Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
    <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
      </EncryptedKey>
    </KeyInfo>
    <CipherData>
      <CipherValue>ydUNqHkMrD...</CipherValue>
    </CipherData>
  </EncryptedData>
</PaymentInfo>
```

XML Encryption in WSS-Core

WSS-Core is a specification under development by OASIS. The specification describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. The message confidentiality is realized by encryption based on XML Encryption.

The WSS-Core specification allows encryption of any combination of body blocks, header blocks, their sub-structures, and attachments of a SOAP message. The specification also requires that when you encrypt parts of a SOAP message, you must prepend a reference from the security header block to the encrypted parts of the message. The reference can be a clue for a recipient to identify which encrypted parts of the message to decrypt.

The XML syntax of the reference varies according to what information is encrypted and how it is encrypted. For example, suppose that the `CreditCard` element in example 4 is encrypted with either a common secret key or the public key of the recipient.

Example 4: Sample SOAP message

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <CreditCard Limit='5,000' Currency='USD'>
        <Number>4019 2445 0277 5567</Number>
        <Issuer>Example Bank</Issuer>
        <Expiration>04/02</Expiration>
      </CreditCard>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The resulting SOAP messages are shown in examples 5 and 6. In these example, the `ReferenceList` and `EncryptedKey` elements are used as references, respectively.

Example 5: SOAP message encrypted with a common secret key

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <ReferenceList xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <DataReference URI='#ed1'/>
      </ReferenceList>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>John Smith</KeyName>
```

```
        </KeyInfo>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example 6: SOAP message encrypted with public key of the recipient

```
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'
  xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'>
  <SOAP-ENV:Header>
    <Security SOAP-ENV:mustUnderstand='1'
      xmlns='http://schemas.xmlsoap.org/ws/2003/06/secext'>
      <EncryptedKey xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
        <KeyInfo xmlns='http://www.w3.org/2000/09/xmldsig#'>
          <KeyName>Sally Doe</KeyName>
        </KeyInfo>
        <CipherData>
          <CipherValue>yMTEyOTA1M...</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI='#ed1'/>
        </ReferenceList>
      </EncryptedKey>
    </Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PaymentInfo xmlns='http://example.org/paymentv2'>
      <Name>John Smith</Name>
      <EncryptedData Id='ed1'
        Type='http://www.w3.org/2001/04/xmlenc#Element'
        xmlns='http://www.w3.org/2001/04/xmlenc#'>
        <EncryptionMethod
          Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
        <CipherData>
          <CipherValue>ydUNqHkMrD...</CipherValue>
        </CipherData>
      </EncryptedData>
    </PaymentInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Relationship to Digital Signature

The WSS-Core specification also provides message integrity, which is realized by digital signature based on XML-Signature.

CAUTION:
A combination of encryption and digital signature over common data introduces cryptographic vulnerabilities.

Securing Web services using XML encryption

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) encryption is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature

- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

XML encryption enables you to encrypt an XML element, the content of an XML element, or arbitrary data such as a XML document. Like XML digital signature, a message is sent by the client as the request sender to the server as the request receiver. The response is sent by the server as the response sender to the client as the request receiver. Unlike XML digital signature, which verifies the authenticity of the sender, XML encryption scrambles the message content using a key, which can be unscrambled by a receiver that possess the same key. You can use XML encryption in conjunction with XML digital signature; scrambling the content while verifying the authenticity of the message sender. To use XML encryption to secure Web services, complete the following tasks. You must use the WebSphere Application Server Toolkit, which is available at the following Web site: http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en

1. Specify the encryption settings for the request sender. The message parts and encryption method settings chosen for the request sender on the client must match the message parts and method settings chosen for the request receiver on the server. To specify the encryption settings for the request sender, complete the following steps:
   a. Configure the client for request encryption: encrypting the message parts.
   b. Configure the client for request encryption: choosing the encryption method.
2. Specify the encryption settings for the request receiver.

   Remember: The decryption settings chosen for the request receiver must match the encryption settings chosen for the request sender.

   To specify the decryption settings for the request receiver, complete the following steps:
   a. Configure the server for request decryption: decrypting message parts.
   b. Configure the server for request decryption: choosing the decryption method.
3. Specify the encryption settings for the response sender. The message parts and encryption method settings chosen for the response sender on the server must match the message parts and method settings chosen for the response receiver on the client. To specify the encryption settings for the response sender, complete the following steps:
   a. Configure the server for response encryption: encrypting message parts.
   b. Configure the server for response encryption: choosing the encryption method.
4. Specify the encryption settings for the response receiver.

   Remember: The decryption settings chosen for the response receiver must match the encryption settings chosen for the response sender.

   To specify the decryption settings for the response receiver, complete the following steps:
   a. Configure the client for response decryption: decrypting message parts.

b. Configure the client for response decryption: choosing the decryption method.

After completing these steps, you have secured your Web services using XML encryption.

Encryption information collection

Use this page to specify the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message, including the body and user name token.

To view this administrative console page, complete the following steps:

1. Click **Enterprise Applications>** *application_name*.
2. Under Related Items, Click **Web Module**.
3. Under Additional Properties, click **Web Services: Server Security Bindings**.
4. Under Request Receiver Binding, click **Edit > Encryption Information**.
5. Click **New** to create an encryption method.
6. Click **Delete** to delete an encryption method.

Encryption Information:

Specifies the name of the encryption information.

Encryption information configuration settings

Use this page to configure the encryption and decryption parameters. You can use these parameters to encrypt and decrypt various parts of the message including the body and user name token.

To view this administrative console page, click **Applications > Enterprise Applications >***application_name*. Under Related Items, click **Web Module >** *URI_file_name* **> Web Services: Server Security Bindings**. Under Request Receiver Binding, click **Edit > Encryption Information > New**.

Encryption Information Name:

Specifies the name for the encryption information.

Key Locator Reference:

Specifies the name used to reference the key locator.

To specify key locator references, click **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Encryption Key Name:

Specifies the name of the encryption key, which is resolved to the actual key by the specified key locator.

Key Encryption Algorithm:

Specifies the algorithm Uniform Resource Identifier (URI) of the key encryption method.

The following algorithms are supported:

- `http://www.w3.org/2001/04/xmlenc#rsa-1_5`

- http://www.w3.org/2001/04/xmlenc#kw-tripledes

Data Encryption Algorithm:

Specifies the algorithm URI of the data encryption method.

The following algorithm is supported:
- http://www.w3.org/2001/04/xmlenc#tripledes-cbc

Encryption information configuration settings

Use this page to configure the encryption and decryption parameters.
The specifications listed on this page for the signature method, digest method, and canonicalization method are located in the World Wide Web Consortium (W3C) document entitled, *XML Encryption Syntax and Processing: W3C Recommendation 10 Dec 2002*.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >** *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit > Encryption Information**.
4. If the encryption information is not available, select **None**.
5. If the encryption information is available, select **Dedicated Encryption Information**.

Then, specify the configuration in the following fields:

Encryption Information Name:

Specifies the name for the encryption information.

Key Locator Reference:

Specifies the name used to reference the key locator.

To specify key locator references, click **Servers > Application Servers >***server_name*. Under Additional Properties, click **Web Services: Default bindings for Web Services Security > Key Locators**.

Encryption Key Name:

Specifies the name of the encryption key, which is resolved to the actual key by the specified key locator.

Key Encryption Algorithm:

Specifies the algorithm URI of the key encryption method.

The following algorithms are supported:
- http://www.w3.org/2001/04/xmlenc#rsa-1_5
- http://www.w3.org/2001/04/xmlenc#kw-tripledes

By default the Java Cryptography Extension (JCE) is shipped with restricted or limited strength ciphers. To use 192-bit and 256- bit Advanced Encryption Standard (AES) encryption algorithms, you must apply unlimited jurisdiction policy files.

Before downloading these policy files, back up the existing policy files (local_policy.jar and US_export_policy.jar in the jre/lib/security/ directory) prior to overwriting them in case you want to restore the original files later. To download the policy files, complete either of the following sets of steps:

- For WebSphere Application Server platforms using IBM Developer Kit, Java Technology Edition Version 1.4.1, including the AIX, Linux, and Windows platforms, you can obtain unlimited jurisdiction policy files by completing the following steps:
  - Go to the following Web site: http://www.ibm.com/developerworks/java/jdk/security/index.html
  - Click **Policy Files**.

    The unrestrict.jar file is downloaded onto your machine.
- For WebSphere Application Server platforms using the Sun-based Java Development Kit (JDK) Version 1.4.1, including the Solaris environments and the HP-UX platform, you can obtain unlimited jurisdiction policy files by completing the following steps:
  - Go to the following Web site: http://java.sun.com/j2se/1.4.1/download.html
  - Go to the bottom of the Web page and click Download, which is next to Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 1.4.1. The jce_policy-1_4_1.zip file is downloaded onto your machine.

After following either of these sets of steps, two Java Archive (JAR) files are placed in the JVM jre/lib/security/ directory.

Data Encryption Algorithm:

Specifies the algorithm Uniform Resource Identifiers (URI) of the data encryption method.

The following algorithm is supported:

- http://www.w3.org/2001/04/xmlenc#tripledes-cbc

By default the JCE is shipped with restricted or limited strength ciphers. To use 192-bit and 256- bit AES encryption algorithms, you must apply unlimited jurisdiction policy files.

Login bindings configuration settings

Use this page to configure the encryption and decryption parameters.
The pluggable token uses the Java Authentication and Authorization Service (JAAS) CallBackHandler (javax.security.auth.callback.CallBackHandler) interface to generate the token that is inserted into the message. The following list describes the CallBack support implementations:

com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback
: This implementation is used for generating binary tokens inserted as <wsse:BinarySecurityToken/@ValueType> in the message.

javax.security.auth.callback.NameCallback and javax.security.auth.callback.NameCallback
: This implementation is used for generating user name tokens inserted as <wsse:UsernameToken> in the message.

com.ibm.wsspi.wssecurity.auth.callback.XMLTokenSenderCallback
: This implementation is used to generate eXtensible Markup Language (XML) tokens and is inserted as the <SAML: Assertion> element in the message.

com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback
This implementation is used to obtain properties specified in the binding file.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Module >***URI_file_name* > **Web Services: Client Security Bindings** .
3. Under Request Sender Bindings, click **Edit > Login Binding**.

If the encryption information is not available, select **None**.

If the encryption information is available, select **Dedicated Login Binding** and specify the configuration in the following fields:

Authentication Method:

Specifies the unique name for the authentication method.

Callback Handler:

Specifies the name of the callback handler.The callback handler must implement the javax.security.auth.callback.CallbackHandler interface.

Basic Auth User ID:

Specifies the user name for basic authentication. The Basic Auth authentication method provides the capability to define a user ID and password in the binding file.

Basic Auth Password:

Specifies the password for basic authentication.

Token Type URI:

Specifies the Uniform Resource Identifiers (URI) for the token type. This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the XML token `<SAML: Assertion>`.

Token Type Local Name:

Specifies the local name for the token type. This information is inserted as `<wsse:BinarySecurityToken>/ValueType` for the XML token `<SAML: Assertion>`.

Request sender

The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and bindings, located in the `ibm-webservicesclient-bnd.xmi file`. These constraints and bindings apply both to J2EE application clients or when Web services is acting as a client. The security handler acts on the security constraints before sending the SOAP message. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token.
The security handler on the request sender side of the SOAP message enforces the security constraints, located in the `ibm-webservicesclient-ext.xmi` file, and bindings, located in the `ibm-webservicesclient-bnd.xmi file`. These constraints

and bindings apply both to J2EE application clients or when Web services is acting as a client. The security handler acts on the security constraints before sending the SOAP message. For example, the security handler might digitally sign the message, encrypt the message, create a time stamp, or insert a security token. You can specify the following security requirements for the request sender and apply them to the SOAP message:

Integrity (digital signature)
You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:
- Body
- Time stamp
- Security token

Confidentiality (encryption)
You can select multiple parts of a message to be encrypted. The following is a list of confidentiality options:
- Body content
- Username token

Security token
You can insert only one token into the message. The following is a list of security token options:
- Basic authentication, which requires both a user name and password
- Identity assertion, which requires a user name only
- X.509 binary security token
- LTPA binary security token
- Custom token , which is pluggable and allows custom-defined tokens to be inserted into the SOAP message

Timestamp
You can have a time stamp for indicating the freshness of the message.
- Timestamp

Note: Request sender security constraints apply to the SOAP message and must match the security constraint requirements defined in request receiver.

Request sender binding collection:

Use this page to specify the binding configuration to send request messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Client Security Bindings**.
3. Under Request Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of the message including the body and time stamp.

You can also use these parameters for X.509 validation when *Authentication Method* is `IDAssertion` and *ID Type* is `X509Certificate` in the server-level configuration. In such cases, you must fill in the *Certificate Path* fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. Encryption information is used for encrypting and decrypting various parts of a message including the body and user name token.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Login Mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, use the administrative console and click **Security > JAAS Configuration**.

Configuring the client for request encryption: Encrypting the message parts

Prior to completing these steps, read either of the following topics to familiarize yourself with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which message parts to encrypt when configuring the client for request encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand **Request Sender Configuration > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality

reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the message is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting , see XML encryption.

8. Select the parts of the message that you want to encrypt by clicking **Add**. You can select one of the following parts:

   Bodycontent
   User data portion of the message

   Usernametoken
   Basic authentication information, if selected

Once you have specified which message parts to encrypt, you must specify which method to use to encrypt the request message. See Configuring the client for reqiuest encryption: Choosing the encryption method for more information.

Configuring the client for request encryption: Choosing the encryption method

Prior to completing these steps, read either of the following topics to familiarize yourself with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which encryption method to use when configuring the client for request encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand **Security Request Sender Binding Configuration > Encryption Information**.
8. Select an encryption option and click **Edit** to view the encryption information or click **Add** to add another option. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   The encryption name refers to the name of the encryption information entry.

Data encryption method algorithm
The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.

Key encryption method algorithm
The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys.

Encryption key name
The encryption key name represents a *Subject* (*Owner* field of the certificate) from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

Note: The key chosen must be a public key of the target. Encryption must be done using the public key and decryption must be done by the target using the private key (the personal certificate of the target).

Encryption key locator
The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

The encryption key name chosen must refer to a public key of the target. For the encryption key name, use the Subject (Owner field of the certificate) of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the request message to encrypt. See Configuring the client for request encryption: Encrypting the message parts if you have not previously specified this information.

Request receiver

The security handler on the request receiver side of the SOAP message enforces the security specifications defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`) and bindings (`ibm-webservices-bnd.xmi`). The request receiver defines the security requirement of the SOAP message. If the incoming SOAP message does not meet all the security requirements defined, then the request is rejected with the appropriate fault code returned to the sender. For security tokens, the token is validated using Java Authentication and Authorization Service (JAAS) login configuration and authenticated identity is set as the identity for the downstream invocation.
The security handler on the request receiver side of the SOAP message enforces the security specifications defined in the IBM extension deployment descriptor (`ibm-webservices-ext.xmi`) and bindings (`ibm-webservices-bnd.xmi`). The request receiver defines the security requirement of the SOAP message. If the incoming SOAP message does not meet all the security requirements defined, then the request is rejected with the appropriate fault code returned to the sender. For security tokens, the token is validated using Java Authentication and Authorization Service (JAAS) login configuration and authenticated identity is set as the identity for the downstream invocation.

For example, if there is a security requirement to have the SOAP body digitally signed by Joe Smith and if the SOAP body of the incoming SOAP message is not signed by Joe Smith, then the request is rejected.

You can define the following security requirements for request receiver:

Required integrity (digital signature)
You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:
- Body
- Time stamp
- Security token

Required confidentiality (encryption)
You can select multiple parts of a message to be encrypted. The following is a list of confidentiality options:
- Body content
- Token

You can have multiple security tokens. The following is a list of security token options:
- Basic authentication, which requires both a user name and password
- Identity assertion, which requires a user name only
- X.509 binary security token
- LAP binary security token
- Custom token, which is pluggable and allows custom-defined tokens to be validated by the JAAS login configuration

Received time stamp
You can have a time stamp for checking the freshness of the message.
- Time stamp

Note: The security constraint for request sender must match the security requirement of the request receiver for the request to be accepted by the server.

Request receiver binding collection:

Use this page to specify the binding configuration to receive request messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Request Receiver Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of a message including the body, time stamp, and user name token.

You also can use these parameters for X.509 certificate validation when Authentication Method is IDAssertion and ID Type is X509Certificate in the server-level configuration. In such cases, you must fill in the **Certificate Path** fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. This configuration is used to encrypt and decrypt parts of the message that include the body and user name token.

Trust Anchors:

Specifies a list of keystore objects that contain the trusted root certificates that are issued by a certificate authority (CA).

The certificate authority authenticates a user and issues a certificate. The CertPath API uses the certificate to validate the certificate chain of incoming, X.509-formatted security tokens or trusted, self-signed certificates.

Collection Certificate Store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates.The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Trusted ID Evaluators:

Specifies a list of trusted ID evaluators that determine whether to trust the identity-asserting authority or message sender.

The trusted ID evaluators are used to authenticate additional identities from one server to another server. For example, a client sends the identity of user A to server 1 for authentication. Server 1 calls downstream to server 2, asserts the identity of user A, and includes the user ID and password of server 1. Server 2 attempts to establish trust with server 1 by authenticating its user ID and password and checking the trust based on the TrustedIDEvaluator implementation. If the authentication process and the trust check are successful, server 2 trusts that server 1 authenticated user A and a credential is created for user A on server 2 to invoke the request.

Login Mappings:

Specifies a list of configurations for validating tokens within incoming messages.

Login mappings map the authentication method to the Java Authentication and Authorization Service (JAAS) configuration.

To configure JAAS, use the administrative console and click **Security > JAAS Configuration**.

Configuring the server for request decryption: decrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete this task to specify which parts of the request message must be decrypted by the server. You must know which parts of the request message the client encrypts because the server must decrypt the same message parts.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Required Confidentiality** section.
8. Select the parts of the message to decrypt The message parts selected for the request decryption on the server must match the message parts selected for the message encryption on the client. Click **Add** and select either of the following message parts:

   `bodycontent`
   : The user data section of the message.

   usernametoken
   : This token is the basic authentication information.

Once you have specified which parts of the request message to decrypt, you must specify the method used to decrypt the message. See Configuring the server for request decryption: choosing the decryption method for more information.

Configuring the server for request decryption: choosing the decryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete this task to specify which decryption method is used by the server to decrypt the request message. You must know which decryption method the client uses because the server must use the same method.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Encryption Information** section.
8. Click **Edit** to view the encryption information. The following table describes the purpose for each of these selections. Some definitions take from the XML-Encryption specification , which can be found at: http://www.w3.org/TR/xmlenc-core

   Encryption name
   Encryption name is the name of this encryption information entry. This is an alias for the entry.

   Data encryption method algorithm
   Data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Key encryption method algorithm
   Key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys. This algorithm must be the same as the algorithm selected in the client request sender configuration.

   Encryption key name
   Encryption key name represents a Subject (from a certificate) found by the encryption key locator. the Subject is used by the key encryption method algorithm to decrypt the secret key, and the secret key is used to decrypt the data.

   Attention: The key chosen here should be a private key in the keystore configured by the key locator. The key should have the same Subject used by the client to encrypt the data. Encryption must be done using the public key and decryption by the private key (personal certificate). To ensure that the client encrypts the data with the correct public or private key, you must extract the public key from the server keystore and add it to the keystore specified in the encryption configuration information for the client request sender.

   For example, the personal certificate of a server is `CN=Bob, O=IBM, C=US`. Therefore the server contains the public and private key pair. The client sending the request should encrypt the data using the public key for `CN=Bob, O=IBM, C=US`. The server decrypts the data using the private key for `CN=Bob, O=IBM, C=US`.

Encryption key locator
: This represents a reference to a key locator implementation. For more information on configuring key locators, go to the following sections: Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

It is important to note that for decryption, the encryption key name chosen must refer to a personal certificate that can be located by the key locator of the server referenced in the encryption information. Enter the Subject of the personal certificate here, which is typically a Distinguished Name (DN). The Subject uses the default key locator to find the key. If a custom key locator is written, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that finds the correct key store where this alias and certificate exist. Refer to Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console for more information.

You must specify which parts of the request message to decrypt. See Configuring the server for request decryption: decrypting the message parts if you have not previously specified this information.

Response sender

The response sender defines the security requirements of the SOAP response message. The security handler acts on the security constraints defined for the response in the IBM extension deployment descriptors, located in the `ibm-webservices-ext.xmi` file and bindings, located in the `ibm-webservices-bnd.xmi` file. The security handler signs, encrypts, or generates the time stamp for the SOAP response message before the response is send to the caller.
The response sender defines the security requirements of the SOAP response message. The security handler acts on the security constraints defined for the response in the IBM extension deployment descriptors, located in the `ibm-webservices-ext.xmi` file and bindings, located in the `ibm-webservices-bnd.xmi` file. The security handler signs, encrypts, or generates the time stamp for the SOAP response message before the response is send to the caller.

Integrity constraints (digital signature)
: You can select multiple parts of the message to be digitally signed.
- Body
- Time stamp

Confidentiality (encryption)
: You can encrypt the body content of the message

Time stamp
: You can have a time stamp for checking the freshness of the message.

Note: The security constraints that apply to the SOAP response message must match the security requirements defined in the response receiver. Otherwise, the response is rejected by the response receiver (caller).

Response sender binding collection:

Use this page to specify the binding configuration for sender response messages for Web services security.

To view this administrative console page, complete the following steps:

1. Click **Applications > Enterprise Applications >***application_name*.

2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters.

You also can use these parameters for X.509 certificate validation when Authentication Method is IDAssertion and ID Type is X509Certificate in the server-level configuration. In such cases, you must fill-in the **Certificate Path** fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a keystore file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Configuring the server for response encryption: encrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which parts of the response message to encrypt when configuring the server for response encryption:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand **Response Sender Service Configuration Details > Confidentiality**. Confidentiality refers to encryption while integrity refers to digital signing. Confidentiality reduces the risk of someone being able to understand the message flowing across the Internet. With confidentiality specifications, the response is encrypted before it is sent and decrypted when it is received at the correct target. For more information on encrypting, see XML encryption.

8. Select the parts of the response that you want to encrypt by clicking **Add** and selecting **Bodytoken** or **Usernametoken**. The following information describes the message parts:

   Bodycontent
   User data portion of the message.

   Usernametoken
   Basic authentication information, if selected.

   A user name token does not appear in the response. You do not need to select this option for the response. If you select this option, make sure that you also select it for the client response receiver. If you do not select it, make sure that you do not select it for the client response receiver either.

Once you have specified which message parts to encrypt, you must specify which method is used to encrypt the message. See Configuring the server for response encryption: choosing the encryption method for more information.

Configuring the server for response encryption: Choosing the encryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Binding Configurations** tab in the Web Services Editor within the Assembly Toolkit:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

These two tabs are used to configure the Web Services Security extensions and Web Services Security bindings, respectively.

Complete the following steps to specify which method the server uses to encrypt the response message:

1. Launch the Assembly Toolkit.
2. Click **Windows > Open Perspective > J2EE** to access the Assembly Toolkit perspective.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand **Response Sender Binding Configuration Details > Encryption Information**.
8. Click **Edit** to view the encryption information. The following table describes the purpose of this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   The encryption name refers to the name of the encryption information entry.

   Data encryption method algorithm
   The data encryption method algorithms are designed for encrypting

and decrypting data in fixed size, multiple octet blocks. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

Key encryption method algorithm
The key encryption method algorithms are public key encryption algorithms that are specified for encrypting and decrypting keys. The algorithm selected for the server response sender configuration must match the algorithm selected in the client response receiver configuration.

Encryption key name
The encryption key name represents a Subject from a certificate found by the encryption key locator, which is used by the key encryption method algorithm to encrypt the private key. The private key is used to encrypt the data.

Important: The key name chosen in the server response sender encryption information must be the public key of the key configured in the client response receiver encryption information. Encryption by the response sender must be done using the public key and decryption must be done by the response receiver using the associated private key (the personal certificate of the response receiver).

Encryption key locator
The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

The encryption key name chosen must refer to a public key of the response receiver. For the encryption key name, use the Subject of the public key certificate, typically a Distinguished Name (DN). The name chosen is used by the default key locator to find the key. If you write a custom key locator , the encryption key name might be anything used by the key locator to find the correct encryption key (a public key). The encryption key locator references the implementation class that finds the correct key store where the alias and certificate exist. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the response message to encrypt. See Configuring the server for response encryption: encrypting the message parts if you have not previously specified this information.

Response receiver

The response receiver defines the security requirements of the response received from a request to a Web service. The security handler enforces the security constraints based on the security requirements defined in the IBM extension deployment descriptor, located in the `ibm-webservicesclient-ext.xmi` file and in the bindings, located in the `ibm-webservicessclient-bnd.xmi` file.
The response receiver defines the security requirements of the response received from a request to a Web service. The security handler enforces the security constraints based on the security requirements defined in the IBM extension deployment descriptor, located in the `ibm-webservicesclient-ext.xmi` file and in the bindings, located in the `ibm-webservicessclient-bnd.xmi` file.

For example, the security requirement might be to have the response SOAP body encrypted. If the SOAP body of the SOAP message is not encrypted, the response is rejected and the appropriate fault code is communicated back to the caller of the Web services.

You can specify the following security requirements for response receiver:

Required integrity (digital signature)
: You can select multiple parts of a message to be digitally signed. The following is a list of integrity options:
  - Body
  - Time stamp

Required confidentiality (encryption)
: You can encrypt the body content of the message.

Received time stamp
: You can have a time stamp for checking the freshness of the message.

Note: The security constraints for response sender must match the security requirements of the response receiver. If the constraints do not match, the response is not accepted by the caller or sender.

Response receiver binding collection:

Use this page to specify the binding configuration for receiver response messages for Web services security.

To view this administrative console page, complete the following steps:
1. Click **Applications > Enterprise Applications >***application_name*.
2. Under Related Items, click **Web Modules** > *URI_file_name* > **Web Services: Server Security Bindings**.
3. Under Response Sender Binding, click **Edit**.

Signing Information:

Specifies the configuration for the signing parameters. Signing information is used to sign and validate parts of the message including the body and time stamp.

You can also use these parameters for X.509 validation when *Authentication Method* is `IDAssertion` and *ID Type* is `X509Certificate` in the server-level configuration. In such cases, you must fill in the *Certificate Path* fields only.

Encryption Information:

Specifies the configuration for the encrypting and decrypting parameters. Encryption information is used for encrypting and decrypting various parts of a message including the body and user name token.

Trust Anchors:

Specifies a list of keystore objects that contain the trusted root certificates, that are self-signed or issued by a certificate authority.

The certificate authority authenticates a user and issues a certificate. After the certificate is issued, the key store objects, which contain these certificates, use the certificate for certificate path or certificate chain validation of incoming X.509-formatted security tokens.

Collection Certificate Store:

Specifies a list of the untrusted, intermediate certificate files.

The collection certificate store contains a chain of untrusted, intermediate certificates.The CertPath API attempts to validate these certificates, which are based on the trust anchor.

Key Locators:

Specifies a list of key locator objects that retrieve the keys for digital signature and encryption from a key store file or a repository. The key locator maps a name or logical name to an alias or maps an authenticated identity to a key. This logical name is used to locate a key in a key locator implementation.

Configuring the client for response decryption: decrypting the message parts

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which response message parts to decrypt when configuring the client for response decryption. The server response encryption and client response decryption configurations must match.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Response Receiver Configuration > Required Confidentiality** section.
8. Select the parts of the message that you must decrypt by clicking **Add** and selecting either **Bodycontent** or **Usernametoken**. The following information describes these message parts:

   Bodycontent
   : The user data portion of the message.

Usernametoken
The basic authentication information, if selected.

The information selected in this step is encrypted by the server in the response sender.

Important: A username token is typically not sent in the response. Thus, you usually do not need to select username token.

Once you have specified which message parts to decrypt, you must specify which method to use when decrypting the response message. See Configuring the client for response decryption: choosing a decryption method for more information.

Configuring the client for response decryption: choosing a decryption method

Prior to completing these steps, read either of the following topics to become familiar with the **Security Extensions** tab and the **Port Binding** tab in the Web Services Client Editor within the Assembly Toolkit:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

These two tabs are used to configure the Web services security extensions and Web services security bindings, respectively.

Complete the following steps to specify which decryption method to use when the client decrypts the response message. The server response encryption and client response decryption configurations must match.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Response Receiver Binding Configuration > Encryption Information** section. For more information on encrypting and decrypting SOAP messages, see XML encryption.
8. Click **Edit** to view the encryption information. The following table describes the purpose for this information. Some of these definitions are based on the XML-Encryption specification, which is located at the following address: http://www.w3.org/TR/xmlenc-core

   Encryption name
   The encryption name refers to the alias used for the encryption information entry.

   Data encryption method algorithm
   The data encryption method algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks.

Key encryption method algorithm
: The key encryption method algorithms are public key encryption algorithms specified for encrypting and decrypting keys.

Encryption key name
: The encryption key name represents a Subject from a certificate found by the encryption key locator. The Subject is used by the key encryption method algorithm to decrypt the secret key. The secret key is used to decrypt the data.

  Important: The key chosen must be a private key of the client. Encryption must be done using the public key and decryption must be done by the private key (personal certificate). For example, the personal certificate of the client is: `CN=Alice, O=IBM, C=US`. Therefore, the client contains the public and private key pair. The target server that sends the response encrypts the secret key using the public key for `CN=Alice, O=IBM, C=US`. The client decrypts the secret key using the private key for CN=Alice, O=IBM, C=US.

Encryption key locator
: The encryption key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

For decryption, the encryption key name chosen must refer to a personal certificate that can be located by the client key locator. The Subject (owner field of the certificate) of the personal certificate should be entered in the Encryption key name, this is typically a Distinguished Name (DN). The default key locator uses the Encryption key name to find the key within the keystore. If you write a custom key locator, the encryption key name can be anything used by the key locator to find the correct encryption key. The encryption key locator references the implementation class that locates the correct key store where this alias and certificate exists. For more information, see Configuring key locators using the Assembly Toolkit and Configuring key locators using the Administrative Console.

You must specify which parts of the request message to decrypt. See the topicConfiguring the client for response decryption: decrypting the message parts if you have not previously specified this information.

Securing Web services using basicauth authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the basicauth authentication method, the request sender generates a basicauth security token using a callback handler. The request receiver retrieves the basicauth

security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. Trust is established using user name and password validation. To use basicauth authentication to secure Web services, complete the following tasks:

1. Secure the client for basicauth authentication.
   a. Configure the client for basicauth authentication: specifying the method
   b. Configure the client for basicauth authentication: collecting the authentication information
2. Secure the server for basicauth authentication.
   a. Configure the server to handle basicauth authentication
   b. Configure the server to validate basicauth authentication information

After completing these steps, you have secured your Web services using basicauth authentication.

Configuring the client for basic authentication: Specifying the method

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a GUI prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction. For more information on BasicAuth authentication, see BasicAuth authentication method.

Complete the following steps to specify BasicAuth as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB module or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section. The only valid login configuration choices for a pure client are BasicAuth and Signature.
8. Select **BasicAuth** to authenticate the client using a user ID and password. This user ID and password must be specified in the target user registry. The other choice, Signature, attempts to authenticate the client using the certificate used to digitally sign the message.

For more information on getting started with the Web Services Client Editor within the Assembly Toolkit, see either of the following topics:

- Configuring the client security bindings using the Assembly Toolkit
- Configuring the security bindings on a server acting as a client using the administrative console

Once you have specified BasicAuth as the authentication method, you must specify how to collect the authentication information. See Configuring the client for basic authentication: collecting the authentication information.

BasicAuth authentication method:

When you use the BasicAuth authentication method, the security token that is generated is a `<wsse:UsernameToken>` element with `<wsse:Username>` and `<wsse:Password>` elements.

WebSphere Application Server supports text passwords but not password digest because passwords are not stored and cannot be retrieved from the server. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

BasicAuth token generation

The request sender generates a BasicAuth security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler that is used is specified in the `<LoginBinding>` element of the bindings file, `ibm-webservicesclient-bnd.xmi` . The following callback handler implementations are provided with WebSphere Application Server and can be used with the BasicAuth authentication method:

- `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.StdinPromptCallbackHandler`
- `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

BasicAuth token validation

The request receiver retrieves the BasicAuth security token from the SOAP message and validates it using a JAAS login module. The `<wsse:Username>` and `<wsse:Password>` elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file for a match. WebSphere Application Server provides the `WSLogin` default configuration entry, which is suitable for the BasicAuth authentication method.

Configuring the client for basic authentication: collecting the authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Collection of BasicAuth information can occur in many ways including through a GUI prompt, a standard in (Stdin) prompt, or specified in the bindings, which prevents user interaction. For more information on BasicAuth authentication, see BasicAuth authentication method.

Complete this task to specify the authentication information needed for BasicAuth authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** or **Enable** to view the Login Binding information. The login binding information displays and enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that will occur. Select BasicAuth to use basic authentication.

   Token value type URI and Token value type local name
   When you select BasicAuth, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For BasicAuth authentication, you do not need to enter any information.

   Callback handler
   The callback handler specifies the Java Authentication and Authorization Server (JAAS) callback handler implementation for collecting the BasicAuth information. You can use the following default implementations for the callback handler:

   - `com.ibm.wsspi.wssecurity.auth.callback .StdinPromptCallbackHandler`

     This implementation is used for non-GUI console prompts.
   - `com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler`

     This implementation is used for GUI panel prompts.
   - `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

     This implementation is used when you plan to always enter the user ID and password in the BasicAuth user ID and password section that follows.

   Basic Authentication user ID and Basic Authentication password
   When values for BasicAuth user ID and password are entered, regardless of the default callback handler indicated previously, these

user ID and password values are used to authenticate to the server for the Web services security authentication. If you leave these values blank, use either the GUIPromptCallbackHandler or the StdinPromptCallbackHandler implementation, but only on a pure client. Always fill-in these values for any Web service that acts as a client to another Web service and you want to specify BasicAuth for authentication downstream. If you want the client identity of the originator to flow downstream, configure the Web service client to use either ID assertion or Lightweight Third Party Authentication (LTPA). To configure ID assertion, see Configuring the client for identity assertion authentication: Specifying method,Configuring the client for identity assertion authentication: Collecting the authentication method, Configuring the client for LTPA token authentication: Specifying the LTPA token authentication information, and Configuring the client for LTPA token authentication: Collecting the authentication information.

Property
: This field enables you to enter properties and name and value pairs for use by custom callback handlers. For BasicAuth authentication, you do not need to enter any information. To enter a new property, click **Add** and enter the new property and value.

Attention: There is a basic authentication entry in the **Port Qualified Name Binding Details** section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the **Web services security basic authentication** section overrides the basic authentication information specified in the **Port Qualified Name Binding Details** section for authorizing the Web service.

For a server that acts as a client, do not specify a GUI or non-GUI prompt callback handler. To configure BasicAuth authentication from one Web service to a downstream Web service, select the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHander` implementation and explicitly specify the BasicAuth user ID and password. If you want the client identity of the originator to flow downstream, configure the Web service client to use ID assertion. To configure ID assertion, see Configuring the client for identity assertion authentication: Specifying method and Configuring the client for identity assertion authentication: Collecting the authentication method.

To use the BasicAuth authentication method, you must specify the method in the **Login Config** section of the Assembly Toolkit. See Configuring the client for basicauth authentication: specifying the method if you have not previously specified this information.

Identity assertion authentication method:

When using the Identity Assertion (IDAssertion) authentication method, the security token generated is a `<wsse:UsernameToken>` element that contains a `<wsse:Username>` element.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. These two operations, token generation and token validation, are described in the following sections.

Identity assertion token validation:

The request receiver retrieves the IDAssertion security token from the SOAP message and validates it using a Java Authentication and Authorization Service (JAAS) login module. With identity assertion, special processing is required to establish trust before asserting the identity as the established identity of the thread of execution. This special processing is defined by the `<IDAssertion>` element in the deployment descriptor file, `ibm-webservices-ext.xmi`. If all the validation checks are successful, the asserted identity is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the`<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches the `security.xml` file for a matching configuration name entry. If a match is not found it searches the `wsjaas.conf` file. WebSphere Application Server provides the `system.wssecurity.IDAssertion` default configuration entry, which is suitable for the identity assertion authentication method.

The `<IDAssertion>` element in the `ibm-webservices-ext.xmi` deployment descriptor file specifies the special processing required when using the identity assertion authentication method. The `<IDAssertion>` element is composed of two sub-elements: `<IDType>` and `<TrustMode>`.

The `<IDType>` element specifies the method for asserting the identity. The supported values for asserting the identity are:

- Username
- Distinguished name (DN)
- X.509 certificate

When `<IDType>` is *username*, a username token (for example, Bob) is provided. This user name is mapped to a user in the user registry and is the asserted identity after successful trust validation. When the `<IDType>` is *DN*, a user name token containing a distinguished name is provided (for example, `cn=Bob Smith, o=ibm, c=us`). This DN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation. When the `<IDType>` is *X509Certificate*, a binary security token containing an X509 certificate is provided and the SubjectDN from the certificate (for example, `cn=Bob Smith, o=ibm, c=us`) is extracted. This SubjectDN is mapped to a user in the user registry and this user is the asserted identity after successful trust validation.

The `<TrustMode>` element specifies how the trust authority, or asserting authority, provides trust information. The supported values are:

- Signature
- BasicAuth
- No value specified

When `<TrustMode>` is `Signature`, the signature is validated. Then, the signer (for example, `cn=IBM Authority, o=ibm, c=us`) is mapped to an identity in the user registry (for example, IBMAuthority). To ensure that the asserting authority is trusted, the mapped identity (for example, IBMAuthority) is validated against a list of trusted identities. When the `<TrustMode>` is `BasicAuth`, there is a user name token with a user name and password, which is the user name and password of the asserting authority. The user name and password are validated. If they are successfully validated, that user name (for example, IBMAuthority) is validated against a list of trusted identities. If a value is not specified for `<TrustMode>`, trust is presumed and additional trust validation is not performed. This type of identity assertion is called *presumed trust mode*. Use the presumed trust mode only in an environment where the trust is established using some other mechanism.

If all the validations described previously succeed, the asserted identity (for example, Bob) is set as the identity of the thread of execution. If any of the validations fail, the request is rejected with a SOAP fault exception.

Configuring the server to handle BasicAuth authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Once a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and password supplied is invalid, an exception is thrown and the request ends without invoking the resource. For more information on BasicAuth authentication, see BasicAuth.

Complete the following steps to configure the server to handle BasicAuth authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. The options you can select are:
   - BasicAuth
   - Signature
   - ID assertion
   - LTPA

   .
8. Select **BasicAuth** to authenticate the client using a user ID and password. The client must specify a valid user ID and password in the server user registry.

   Important: You can select multiple login configurations, which means that different types of security information might be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login

configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token. For ID assertion to work properly, list ID assertion ahead of BasicAuth in the processing list or the BasicAuth processing overrides the IDAssertion processing.

Once you have specified how the server will handle BasicAuth authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate basicauth authentication information for more information.

Configuring the server to validate BasicAuth authentication information

BasicAuth refers to the user ID and password of a valid user in the registry of the target server. Once a request is received that contains basic authentication information, the server needs to log in to form a credential. The credential is used for authorization. If the user ID and password supplied is invalid, an exception is thrown and the request ends without invoking the resource. For more information on BasicAuth authentication, see BasicAuth.

Complete the following steps to specify how the server validates the BasicAuth authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **BasicAuth** to use basic authentication.

   Configuration name
   This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the BasicAuth authentication method, enter **WSLogin** for the JAAS login Configuration name.

   Use token valid type
   The **Use token value type** option determines if you want to specify a custom token type. For the default Authentication method selections, you do not need to specify this option.

   Token value type URI and Token value type URI local name
   When you select BasicAuth, you cannot edit the token value type URI and local name values. These values are specified for custom

authentication types. For BasicAuth authentication, you do not need to enter any information for these fields.

Callback handler factory class name
This class name creates a JAAS CallbackHandler implementation that understands the following callbacks:

- `javax.security.auth.callback.NameCallback`
- `javax.security.auth.callback.PasswordCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

Callback handler factory property name and Callback handler factory property value
This property is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For BasicAuth, you do not need to enter any property values.

Login mapping property name and Login mapping property value
This property is used to specify properties for a custom login mapping to use. For the default implementations including BasicAuth, you do not need to enter any property values.

You must specify how the server will handle the BasicAuth authentication method. See Configuring the server to handle BasicAuth authentication information if you have not previously specified this information.

Identity assertion

Identity assertion is a method for expressing the identity of the sender (for example, user name) in a SOAP message. When identity assertion is used as a authentication method, the authentication decision is performed based only on the name of the identity, but not on other information such as passwords and certificates.

Identity assertion is a method for expressing the identity of the sender (for example, user name) in a SOAP message. When identity assertion is used as a authentication method, the authentication decision is performed based only on the name of the identity, but not on other information such as passwords and certificates.

ID type

The Web Services Security implementation in WebSphere Application Server can handle the following three types of identity.

User name
Denotes the user name, such as the one in the local operating system (for example, ″alice″). This name is embedded in the `<Username>` element within the `<UsernameToken>` element.

DN Denotes the distinguished name (DN) for the user, such as ″CN=alice, O=IBM, C=US″. This name is embedded in the `<Username>` element within the `<UsernameToken>` element.

X.509 certificate
Represents the identity of the user as a X.509 certificate instead of a string name. This certificate is embedded in the `<BinarySecurityToken>` element.

Managing trust

The intermediary host in the SOAP message itinerary can assert the initial sender's claimed identity. Two methods (called *trust mode*) are supported for this assertion:

Basic authentication
The intermediary adds its user name and password pair to the message.

Signature
The intermediary digitally signs the `<UsernameToken>` element of the initial sender.

Note: This trust mode does not support the X.509 certificate ID type.

Typical scenario

ID assertion is typically used in the multi-hop environment where the SOAP message passes through one or more intermediary hosts. The intermediary host authenticates the initial sender. The following scenario describes the process:

1. The initial sender sends a SOAP message to the intermediary host with some embedded authentication information. This authentication information might be a user name and password pair and an LTPA token.
2. The intermediary host authenticates the initial sender according to the embedded authentication information.
3. The intermediary host removes the authentication information from the SOAP message and replaces it with the `<UsernameToken>` element, which contains a user name.
4. The intermediary host asserts the trust according to the trust mode.
5. The intermediary host sends the updated SOAP message to the ultimate receiver.
6. The ultimate receiver checks the trust against the intermediary host information according to the configured trust mode. Also, the trusted ID evaluator is invoked.
7. If trust is established by the ultimate receiver, it invokes the Web service under the authorization of the user name (that is, the initial sender) in the SOAP message.

Securing Web services using identity assertion authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the identity assertion authentication method, the security token generates a `<wsee:Username Token>` element that contains a `<wsse:Username>` element. On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, the security token is validated. Unlike basicauth authentication, trust is established through the use of a security token rather than through user name and password validation. To use identity assertion authentication to secure Web services, complete the following tasks:

1. Secure the client for identity assertion authentication.
   a. Configure the client for identity assertion authentication: specifying the method.
   b. Configure the client for identity assertion authentication: collecting the authentication information.
2. Secure the server for identity assertion authentication.
   a. Configure the server to handle identity assertion authentication.
   b. Configure the server to validate identity assertion authentication information.

After completing these steps, you have secured your Web services using identity assertion authentication.

Configuring the client for identity assertion: specifying the method

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a Web service acting as a client to a downstream Web service.

In order for the downstream Web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators.

Complete the following steps to specify identity assertion as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section.
8. Select **IDAssertion** as the authentication method. For more conceptual information on identity assertion authentication, see ID assertion.

9. Expand the **IDAssertion** section.
10. For the ID Type, select **Username**. This works with all registry types and originating authentication methods.
11. For the Trust Mode, select either **BasicAuth** or **Signature**.
    - By selecting **BasicAuth**, you must include basic authentication information (user ID and password), which the downstream Web service has specified in the trusted ID evaluator as a trusted user ID. See Configuring the client for signature authentication: Collecting the authentication information to specify the user ID and password information.
    - By selecting **Signature** , the certificate configured in the **Signature information** section used to sign the data also is used as the trusted subject. The Signature is used to create a credential and the user ID, which the certificate mapped to the downstream registry, is used in the trusted ID evaluator as a trusted user ID.

See Configuring the client security bindings using the Assembly Toolkit for more information on the Web Services Client Editor within the Assembly Toolkit.

Once you have specified identity assertion as the authentication method used by the client, you must specify how to collect the authentication information. See Configuring the client for identity assertion authentication: collecting the authentication information for more information.

Configuring the client for identity assertion: Collecting the authentication method

This task is used to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client. Identity assertion works only when you configure on the client-side of a Web service acting as a client to a downstream Web service.

In order for the downstream Web service to accept the identity of the originating client (just the user name), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators.

Complete the following steps to specify how the client collects the authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.

7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** to view the login binding information and select **IDAssertion**. The login binding dialog displays and either select or enter the following information:

   Authentication method
   : The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use identity assertion.

   Token value type URI and Token value type Local name
   : When you select IDAssertion, you cannot edit the token value type URI and the local name. These values are specifically for custom authentication types. For IDAssertion authentication, you do not need to enter any information.

   Callback handler
   : The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the BasicAuth information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler` implementation for IDAssertion.

   Basic authentication User ID and Basic authentication Password
   : If the trust mode entered in the extensions is BasicAuth, you must specify the trusted user ID and password in these fields. The user ID specified must be an ID that is trusted by the downstream Web service. The Web service trusts the user ID if it is entered as a trusted ID in a trusted ID evaluator in the downstream Web service bindings. If the trust mode entered in the extensions is Signature, you do not need to specify any information in this field.

   Property Name and Property Value
   : This field enables you to enter properties and name and value pairs, for use by custom callback handlers. For IDAssertion, you do not need to specify any information in this field.

To use the identity assertion authentication method, you must specify the method in the **Security Extentions** section of the Assembly Toolkit. See Configuring the client for identity assertion authentication: specifying the method if you have not previously specified this information.

Configuring the server to handle identity assertion authentication

Use this task to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client.

For the downstream Web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. Once it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to configure the server to handle identity assertion authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. The options you can select are:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**
   - **LTPA**
8. Select **IDAssertion** to authenticate the client using the identity assertion data provided. This user ID of the client must be in the target user registry configured in WebSphere Application Server global security. You can select global security in the Administrative Console by clicking **Security > Global security**.

   You can select multiple login configurations, which means that different types of security information can be received at the server. The order in which the login configurations are added decides the order in which they are processed when a request is received. This can cause problems if you have multiple login configurations added that have security tokens in common. For example, ID assertion contains a BasicAuth token, which is the token that is being trusted. For ID assertion to work properly, you must list ID assertion ahead of BasicAuth in the list or BasicAuth processing overrides ID assertion processing.
9. Expand the **IDAssertion** section. You need to select both the **ID Type** and **Trust Mode**.
   a. For **ID Type**, the options are:
      - **Username**
      - **Distinguished name (DN)**
      - **X509certificate**

      These choices are just preferences and are not guaranteed. Most of the time **Username** is used. You must choose the same **ID Type** as the client.
   b. For **Trust Mode**, the options are:
      - **BasicAuth**
      - **Signature**

      The **Trust Mode** refers to the information sent by the client as the trusted ID.
      1) If you select **Signature**, the client signing certificate is sent. This certificate must be mappable to the configured user registry. For **Local OS**, the common name (CN) of the distinguished name (DN) is mapped to a user ID in the registry. For **LDAP**, the DN is mapped to the registry for the ExactDN mode. If it is in the CertificateFilter mode,

attributes are mapped accordingly. In addition, the user name from the credential generated must be in the Trusted ID Evaluator trust list.

2) If you select **BasicAuth**, the client sends basic authentication data (user ID and password). This basicauth data is authenticated to the configured user registry. Once the authentication occurs successfully, the user ID must be part of the trusted ID evaluator trust list.

For more information on getting started with the Web Services Editor within the Assembly Toolkit , see Configuring the server security bindings using the Assembly Toolkit.

Once you have specified how the server will handle identity assertion authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate identity assertion authentication information for more information.

Configuring the server to validate identity assertion authentication information

Use this task to configure identity assertion authentication. The purpose of identity assertion is to assert the authenticated identity of the originating client from a Web service to a downstream Web service. Do not attempt to configure identity assertion from a pure client.

For the downstream Web service to accept the identity of the originating client (user name only), you must supply a special trusted BasicAuth credential that the downstream Web service trusts and can authenticate successfully. You must specify the user ID of the special BasicAuth credential in a trusted ID evaluator on the downstream Web service configuration. For more information on trusted ID evaluators, see Trusted ID evaluators. The server side passes the special BasicAuth credential into the trusted ID evaluator, which returns `true` or `false` that this ID is trusted. Once it is trusted, the user name of the client is mapped to the credential, which is used for authorization.

Complete the following steps to validate the identity assertion authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information. Click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method

   The authentication method specifies the type of authentication that occurs. Select **IDAssertion** to use basic authentication.

Configuration name
: This specifies the JAAS login configuration name. For the IDAssertion authentication method, enter **system.wssecurity.IDAssertion** for the Java Authentication and Authorization Service (JAAS) login configuration name.

Use token value type
: The **Use token value type** option determines if you want to specify a custom token type. For the default authentication method selections, you do not need to specify this option.

Token value type URI and Token value type local name
: When you select ID assertion, you cannot edit the **token value type URI** and **local name** values. These values are specifically for custom authentication types. For the ID assertion authentication method, you do not need to enter any information in these fields.

Callback Handler Factory Class name
: This class name creates a JAAS CallbackHandler implementation that understands the following callbacks:

- `javax.security.auth.callback.NameCallback`
- `javax.security.auth.callback.PasswordCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
- `com.ibm.wsspi.wssecurity.auth.callback .XMLTokenReceiverCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

For any of the default Authentication methods (BasicAuth, IDAssertion, and Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including IDAssertion: `com.ibm.wsspi.wssecurity .auth.callback.WSCallbackHandlerFactoryImpl`

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property name and Callback handler factory property value
: This property is used to specify callback handler properties for Custom callback handler factory implementations. The default callback handler factory implementation does not need any properties to be specified. For ID assertion, you do not need to enter any values for this property.

Login mapping property name and Login mapping property value
: This option is used to specify properties for a custom login mapping. For the default implementations including IDAssertion, you do not need to enter any properties for this option.

9. Expand the **Trusted ID Evaluator** section.
10. Click **Edit** to see a dialog displaying all the trusted ID evaluator information. The following table describes the purpose of this information.

Class name
: The class name refers to the implementation of the trusted ID evaluator that you want to use. Enter the default implementation as `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluatorImpl`. If you want

to implement your own trusted ID evaluator, you must implement the `com.ibm.wsspi.wssecurity.id.TrustedIDEvaluator` interface.

Property name
: The name is the name of this configuration. Enter `BasicIDEvaluator` for lack of a better reference.

Property value
: The property defines name and value pairs that can be used by the trusted ID evaluator implementation. For the default implementation, the trusted list is defined here. When a request comes in and the trusted ID is verified, the user ID, as it appears in the user registry, must be listed in this property. Specify the property as a name and value pair where the name is *trunstedId_n*. *n* is an integer starting from 0 and the value is the user ID associated with that name. An example list with the trusted names include two properties. For example: `trustedId_0 = user1`, `trustedId_1 = user2`. The previous example means that both user1 and user2 are trusted. user1 and user2 must be listed in the configured user registry.

11. Expand the **Trusted ID Evaluator Reference** section.
12. Click **Enable** to add a new entry. The text you enter for the **Trusted ID Evaluator Reference** must be the same as the name entered previously in the **Trusted ID Evaluator**. Make sure that the name matches exactly as the information is case sensitive. If an entry is already specified, you can change it by clicking **Edit**.

You must specify how the server will handle the identity assertion authentication method. See Configuring the server to handle identity assertion authentication if you have not previously specified this information.

Securing Web services using signature authentication

WebSphere Application Server provides several different methods to secure your Web services; eXtensible Markup Language (XML) digital signature is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

With the signature authentication method, the request sender generates a signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. To use signature authentication to secure Web services, complete the following tasks:

1. Secure the client for signature authentication.
   a. Configure the client for signature authentication: specifying the method.
   b. Configure the client for signature authentication: collecting the authentication information.
2. Secure the server for signature authentication.

a. Configure the server to handle signature authentication.
b. Configure the server to validate signature authentication information.

After completing these steps, you have secured your Web services using signature authentication.

Configuring the client for signature authentication: specifying the method

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see Signature authentication method.

Complete the following steps to specify signature as the authentication method:

1. Launch the Assembly Toolkit.
2. Click **Windows > Open Perspective > J2EE** to access the Assembly Toolkit perspective.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section. The following valid login configuration options for a managed client and Web services acting as a client are:

   BasicAuth
   You can use this option for a managed client.

   Signature
   You can use this option for a managed client.

   IDAssertion
   You can use this option for Web services acting as a client.
8. Select **Signature** to authenticate the client using the certificate used to digitally sign the request.

For more information on getting started with the Web Services Client Editor within the Assembly Toolkit , see Configuring the client security bindings using the Assembly Toolkit.

Once you have specified signature as the authentication method, you must specify how to collect the authentication information. See Configuring the client for signature authentication: collecting the authentication information for more information.

Signature authentication method:

When using the signature authentication method, the security token is generated with a `<ds:Signature>` and a `<wsse:BinarySecurityToken>` element.

On the request sender side, a callback handler is invoked to generate the security token. On the request receiver side, a Java Authentication and Authorization

Service (JAAS) login module is used to validate the security token. These two operations, token generation and token validation, are described in the following sections.

Signature token generation
The request sender generates a Signature security token using a callback handler. The security token returned by the callback handler is inserted in the SOAP message. The callback handler is specified in the `<LoginBinding>` element of the bindings file, `ibm-webservicesclient-bnd.xmi`. WebSphere Application Server provides the following callback handler implementation that can be used with the Signature authentication method: `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

Security token validation
The request receiver retrieves the Signature security token from the SOAP message and validates it using a JAAS login module. The `<ds:Signature>` and `<wsse:BinarySecurityToken>` elements in the security token are used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. This Subject then is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault exception.

The JAAS login configuration is specified in the `<LoginMapping>` element of the bindings file. There are default bindings specified in the `ws-security.xml` file. However, you can override these bindings using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory and a ConfigName. The CallbackHandlerFactory specifies the name of a class that is used for creating the JAAS CallbackHandler object. WebSphere Application Server provides the `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImp` CallbackHandlerFactory implementation. The ConfigName specifies a JAAS configuration name entry. WebSphere Application Server searches in the `security.xml` file for a matching configuration name entry. If a match is not found, it searches the `wsjaas.conf` file. WebSphere Application Server provides the `system.wssecurity.Signature` default configuration entry, which is suitable for the signature authentication method.

Configuring the client for signature authentication: collecting the authentication information

This task is used to configure signature authentication. A signature refers to the use of an X.509 certificate to login on the target server. For more information on signature authentication, see Signature authentication method.

Complete the following steps to specify how the client collects the authentication information for signature authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.

5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Signing Information** and click **Edit** to modify the signing key name and signing key locator. To create new signing information, click **Enable**. The certificate that is sent to login at the server is the one configured in the Signing Information section. Review the section on Key locators to understand how the signing key name maps to a key within the key locator entry.

   The following list describes the purpose of this information. Some of these definitions are based on the XML-Signature specification, which is located at the following address: http://www.w3.org/TR/xmldsig-core

   Canonicalization method algorithm
   : The canonicalization method algorithm is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.

   Digest mehod algorithm
   : The digest method algorithm is the algorithm applied to the data after transforms are applied, if specified, to yield the `<DigestValue>`. The signing of the DigestValue binds resource content to the signer key. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signature method algorithm
   : The signature method is the algorithm that is used to convert the canonicalized `<SignedInfo>` into the `<SignatureValue>`. The algorithm selected for the client request sender configuration must match the algorithm selected in the server request receiver configuration.

   Signing key name
   : The signing key name represents the key entry associated with the signing key locator. The key entry refers to an alias of the key, which is used to sign the request.

   Signing key locator
   : The signing key locator represents a reference to a key locator implementation. For more information on configuring key locators, see Key locators.
8. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
9. Click **Edit** to view the Login Binding information. The login binding information displays and either select or enter the following information:

   Authentication method
   : The authentication method specifies the type of authentication that occurs. Select **Signature** to use signature authentication.

   Token value type URI and Token value type URI local name
   : When you select **Signature**, you cannot edit the **Token value type URI** and **Local name** values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information.

   Callback handler
   : The callback handler specifies the Java Authentication and

Authorization Server (JAAS) callback handler implementation for collecting signature information. Enter the following callback handler for signature authentication: `com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler`

This callback handler is used because signature does not require user interaction.

Basic authentication User ID and Basic authentication Password
Do not enter anything in the BasicAuth fields when Signature authentication is desired.

Property name and property value
This field enables you to enter properties and name and value pairs for use by custom callback handlers. For signature authentication, you do not need to enter any information.

Important: There is a basic authentication entry in the Port Qualified Name Binding Details section. This entry is used for HTTP transport authentication, which might be required if the router servlet is protected.

Information specified in the Web services security signature authentication section overrides the basic authentication information specified in the Port Qualified Name Binding Details section for authorizing the Web service.

To use the signature authentication method, you must specify the authentication method in the **Login Config** section of the Assembly Toolkit. See Configuring the client for signature authentication: specifying the method if you have not previously specified this information.

Configuring the server to handle signature authentication

This task is used to configure signature authentication at the server. Signature refers to the an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. Once a request is received by the server that contains certificate, the server needs to log in to form a credential. The credential is used for authorization. If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource. For more information on signature authentication, see Signature authentication method.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Config** section. You can select from the following options:
   - **BasicAuth**
   - **Signature**
   - **ID assertion**

- **LTPA**

8. Select **Signature** to authenticate the client using an X509 certificate. The certificate that is sent from the client is the certificate used for signing the message. You must be able to map this certificate to the configured user registry. For Local OS, the common name (cn) of the distinguished name (DN) is mapped to a user ID in the registry. For LDAP, you can configure multiple mapping modes:
   - EXACT_DN is the default mode that directly maps the DN of the certificate to an entry in the LDAP server.
   - CERTIFICATE_FILTER is the mode that allows the the LDAP advanced configuration to have a place to specify a filter that maps specific attributes of the certificate to specific attributes of the LDAP server.

For more information on getting started with the Web Services Editor within the Assembly Toolkit, see Configuring the server security bindings using the Assembly Toolkit.

Once you have specified how the server will handle signature authentication information, you must specify how the server validates the authentication information. See Configuring the server to validate signature authentication information for more information.

Configuring the server to validate signature authentication information

This task is used to configure signature authentication at the server. Signature refers to the an X.509 certificate sent by the client to the server. The certificate is used to authenticate to the user registry configured at the server. Once a request is received by the server that contains certificate, the server needs to log in to form a credential. The credential is used for authorization. If the certificate supplied cannot be mapped to an entry in the user registry, an exception is thrown and the request ends without invoking the resource. For more information on signature authentication, see Signature authentication method.

Complete the following steps to configure the server to validate signature authentication:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the login mapping information or click **Add** to add new login mapping information. The login mapping dialog displays and either select or enter the following information:

   Authentication method

   The authentication method specifies the type of authentication that will occur. Select **Signature** to use signature authentication.

Configuration name
: This specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the signature authentication method, enter `system.wssecurity.Signature` for the JAAS login configuration name. This specification logs in with the `com.ibm.wsspi.wssecurity.auth.module.SignatureLoginModule` JAAS login module.

Use token value type
: This determines if you want to specify a custom token type. For the default Authentication method selections, you don't need to specify this.

URI and local name
: When you select Signature, you cannot edit the token value type URI and local name values. These values are specifically for custom authentication types. For signature authentication, you do not need to enter any information here.

Callback handler factory class name
: This class name creates a JAAS CallbackHandler implementation that understands the following callback handlers:

- `javax.security.auth.callback.NameCallback`
- `javax.security.auth.callback.PasswordCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
- `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

For any of the default Authentication methods (BasicAuth, IDAssertion, Signature), use the callback handler factory default implementation. Enter the following class name for any of the default Authentication methods including signature: `com.ibm.wsspi.wssecurity .auth.callback.WSCallbackHandlerFactoryImpl`.

This implementation creates the correct callback handler for the default implementations.

Callback handler factory property name and callback handler factory property value
: This field is used to specify callback handler properties for custom callback handler factory implementations. You do not need to specify any properties for the default callback handler factory implementation. For signature, you do not need to enter any properties for this field.

Login mapping property name and login mapping property value
: This field is used to specify properties for a custom login mapping to use. For the default implementations including signature, you do not need to enter any properties for this field.

You must specify how the server will handle the signature authentication method. See Configuring the server to handle signature authentication if you have not previously specified this information.

Token type overview

A username token consists of a user name and, optionally, password information. You can include a username token directly in the `<Security>` header within the message. Binary tokens, such as X.509 certificates, Kerberos tickets,, Lightweight

Third-party Authentication (LTPA) tokens, or other non-XML formats, require a special encoding for inclusion. The Web services security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets; and how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message.

The proposed Web services security draft defined two types of security tokens:

- Username token
- Binary security token

A username token consists of a user name and, optionally, password information. You can include a username token directly in the <Security> header within the message. Binary tokens, such as X.509 certificates, Kerberos tickets,, Lightweight Third-party Authentication (LTPA) tokens, or other non-XML formats, require a special encoding for inclusion. The Web services security specification describes how to encode binary security tokens such as X.509 certificates and Kerberos tickets; and how to include opaque encrypted keys. The specification also includes extensibility mechanisms that you can use to further describe the characteristics of the credentials that are included with a message.

WebSphere Application Server, Version 5.0.2 supports user name tokens, which include both user name and password for basic authentication and user name, which are used for identity assertion. The WebSphere Application Server, Version 5.0.2 binary security token implementation supports both X.509 certificates and LTPA binary security. You can extended the implementation to generate other type of tokens. However, Kerberos tickets are not supported in WebSphere Application Server, Version 5.0.2. Each type of token is processed by a corresponding token generation and validation module. The binary token generation and validation modules are pluggable, which is based on the Java Authentication and Authorization Service (JAAS) framework. For example, arbitrary XML-based tokenformat is supported using the JAAS pluggable framework. WebSphere Application Server, Version 5.0.2 does not support an XML-based token that is used in SecurityTokenReference.

You can define the types of tokens that the message can accept in the deployment descriptor extension file, ibm.webservices-ext.xmi. A message receiver might support one or more types of security tokens. The following example shows that the receiver supports four types of security tokens:

```
?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wsext:WsExtension xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wsext="http://www.ibm.com/websphere/appserver/schemas/5.0.2/wsext.xmi"
xmi:id="WsExtension_1052760331306" routerModuleName="StockQuote.war">
  <wsDescExt xmi:id="WsDescExt_1052760331306" wsDescNameLink="StockQuoteFetcher">
    <pcBinding xmi:id="PcBinding_1052760331326" pcNameLink="urn:xmltoday-delayed-quotes"
      scope="Session">
       <serverServiceConfig
         xmi:id="ServerServiceConfig_1052760331326"actorURI="myActorURI">
          <securityRequestReceiverServiceConfig
           xmi:id="SecurityRequestReceiverServiceConfig_1052760331326">
            <loginConfig xmi:id="LoginConfig_1052760331326">
              <authMethods xmi:id="AuthMethod_1052760331326" text="BasicAuth"/>
              <authMethods xmi:id="AuthMethod_1052760331327" text="IDAssertion"/>
              <authMethods xmi:id="AuthMethod_1052760331336" text="Signature"/>
              <authMethods xmi:id="AuthMethod_1052760331337" text="LTPA"/>
           </loginConfig>
<idAssertion xmi:id="IDAssertion_1052760331336" idType="Username" trustMode="Signature"/>
```

The message sender might choose one of the token types that are supported by the receiver when sending a message. You can define the type of token to be used by the sending side in the client descriptor extension file, `ibm-webservicesclient-ext.xmi`. The following example shows that the sender chooses to send a UsernameToken to the receiver:

```
<?xml version="1.0" encoding="UTF-8"?>
<com.ibm.etools.webservice.wscext:WsClientExtension xmi:version="2.0"
mlns:xmi="http://www.omg.org/XMI"
xmlns:com.ibm.etools.webservice.wscext=
   "http://www.ibm.com/websphere/appserver/schemas/5.0.2/wscext.xmi"
xmi:id="WsClientExtension_1052760331496">
<ServiceRefs xmi:id="ServiceRef_1052760331506" serviceRefLink="service/StockQuoteService">
    <portQnameBindings xmi:id="PortQnameBinding_1052760331506"
portQnameLocalNameLink="StockQuote">
      <clientServiceConfig xmi:id="ClientServiceConfig_1052760331506"
actorURI="myActorURI">
          <securityRequestSenderServiceConfig
xmi:id="SecurityRequestSenderServiceConfig_1052760331506" actor="myActorURI">
          <loginConfig xmi:id="LoginConfig_1052760331506" authMethod="BasicAuth"/>
```

Username token

You can use the UsernameToken to propagate a user name and, optionally, password information. Also, you can use this token type to carry basic authentication information. Both a user name and password are used to authenticate the message. A `UsernameToken` containing the user name is used in identity assertion, which establishes the identity of the user based on the trust relationship.

The following example shows the the syntax of the UsernameToken element:

```
<UsernameToken Id="...">
    <Username>...</Username>
    <Password Type="...">...</Password>
</UsernameToken>
```

The Web services security specification defines the following password types:

wsse:PasswordText (default)
: This type is the actual password for the user name.

wsse:PasswordDigest
: The type is the digest of the password for the user name. The value is a base64-encoded SHA1 hash value of the UTF8-encoded password.

WebSphere Application Server supports the default PasswordText type. However, it does not support password digest because most user registry security policies do not expose the password to the application software.

The following illustrates the use of the `<UsernameToken>` element:

```
<S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
    <S:Header>
            ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>Joe</wsse:Username>
                <wsse:Password>ILoveJava</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
    </S:Header>
</S:Envelope>
```

Binary security token

The `ValueType` attribute identifies the type of the security token, for example, an LTPA token. The `EncodingType` indicates how the security token is encoded, for example, Base64Binary. The `BinarySecurityToken` element defines a security token that is binary encoded. The encoding is specified using the `EncodingType` attribute. The value type and space are specified using the `ValueType` attribute. The Web services security implementation for WebSphere Application Server, Version 5.0.2 supports both LTPA and X.509 certificate binary security tokens.
A binary security token has the following attributes that are used to interpret it:

- Value type
- Encoding type

The `ValueType` attribute identifies the type of the security token, for example, an LTPA token. The `EncodingType` indicates how the security token is encoded, for example, Base64Binary. The `BinarySecurityToken` element defines a security token that is binary encoded. The encoding is specified using the `EncodingType` attribute. The value type and space are specified using the `ValueType` attribute. The Web services security implementation for WebSphere Application Server, Version 5.0.2 supports both LTPA and X.509 certificate binary security tokens.

The following is an example of an LTPA binary security token in a Web services security message header:

```
wsse:BinarySecurityToken xmlns:ns7902342339871340177=
    "http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
     EncodingType="wsse:Base64Binary"
     ValueType="ns7902342339871340177:LTPA">
     MIZ6LGPt2CzXBQfio9wZTo1VotWov0NW3Za6lU5K7Li78DSnIK6iHj3hxXgrUn6p4wZI
     8Xg26havepvmSJ8XxiACMihTJuh1t3ufsrjbFQJOqh5VcRvI+AKEaNmnEgEV65jUYAC9
     C/iwBBWk5U/6DIk7LfXcTT0ZPAd+3D3nCS0f+6tnqMou8EG9mtMeTKccz/pJVTZjaRSo
     msu0sewsOKfl/WPsjW0bR/2g3NaVvBy18VlTFBpUbGFVGgzHRjBKAGo+ctkl80nlVLIk
     TUjt/XdYvEpOr6QoddGi4okjDGPyyoDxcvKZnReXww5UsoqlpfXwN4KG9as=
</wsse:BinarySecurityToken></wsse:Security></soapenv:Header>
```

As shown in the example, the token is Base64Binary encoded.

XML token

XML tokens are offered in two formats, Security Assertion Markup Language (SAML) and eXtensible rights Markup Language (XrML).
XML-based security tokens are growing in popularity. Two well-known formats are:

- Security Assertion Markup Language (SAML)
- eXtensible rights Markup Language (XrML)

The extensibility of the `<wsse:Security>` header in XML-based security tokens enables you to directly insert these security tokens into the header.

SAML assertions are attached to Web services security messages using Web Services by placing assertion elements inside the `<wsse:Security>` header. The following example illustrates a Web services security message with a SAML assertion token.

```
S:Envelope xmlns:S="...">&
      <wsse:Security xmlns:wsse="...">
          <saml:Assertion
                    MajorVersion="1"
                    MinorVersion="0"
                    AssertionID="SecurityToken-ef375268"
                       Issuer="elliotw1"
                       IssueInstant="2002-07-23T11:32:05.6228146-07:00"
```

```
                    xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion">
                    ...
          </saml:Assertion>
     </wsse:Security>
 </S:Header>
 <S:Body>
 ...
 </S:Body>
</S:Envelope>
```

For more information on SAML and XrML, see Resources for learning.

Security token

A security token represents a set of claims made by a client that might include a name, password, identity, key, certificate, group, privilege, and so on.

Web services security provides a general-purpose mechanism to associate security tokens with messages for single message authentication. A specific type of security token is not required by Web services security. Web services security is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof that they have a particular business certification.

A security token is embedded in the SOAP message within the SOAP header. The security token within in the SOAP header is propagated from the message sender to the intended message receiver. On the receiving side, the WebSphere Application Server security handler authenticates the security token and sets up the caller identity on the thread of execution.

Securing Web services using a pluggable token

WebSphere Application Server provides several different methods to secure your Web services; a pluggable token is one of these methods. You might secure your Web services using any of the following methods:

- XML digital signature
- XML encryption
- Basicauth authentication
- Identity assertion authentication
- Signature authentication
- Pluggable token

Complete the following steps to secure your Web services using a pluggable token:

1. Generate a security token using the JAAS CallbackHandler interface. The Web services security run time uses the Java Authentication and Authorization Service (JAAS) CallbackHandler interface as a plug-in to generate security tokens on the client side or when Web services is acting as client.
2. Configure your pluggable token. To use pluggable tokens to secure your Web services, you must configure both the client request sender and the server request receiver. You can configure your pluggable tokens using either the WebSphere Application Server administrative console or the WebSphere Application Server Toolkit. For more information, see the following topics:
   - Configuring pluggable tokens using the WebSphere Application Server Toolkit
   - Configuring pluggable tokens using the administrative console

Configuring pluggable tokens using the Assembly Toolkit

Prior to completing these steps, it is assumed that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with a Web Services for J2EE (JSR 109) enterprise application. If not, see Developing Web services to create Web services-enabled J2EE with a JSR 109 enterprise application. See either of the following topics for an introduction of how to manage Web services security binding information for the server:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

Important: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

You must specify the security constraints in the `ibm-webservicesclient-ext.xmi` and `ibm-webservices-ext.xmi` files for the required tokens using the Assembly Toolkit.

Complete the following steps to configure the request sender using the `ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` files:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Package Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab. The Web Service Client Security Extensions editor displays.
   a. Under Service References, select an existing service reference or click **Add** to create a new one.
   b. Under Port Qname Bindings, select an existing port qualified name for the selected service reference or click **Add** to create a new port name binding.
   c. Under Request Sender Configuration: Login Config, select an exiting authentication method or type in a new one in the editable list box (Lightweight Third-Party Authorization (LTPA) is a supported token generation when Web services is acting as client).
   d. Click **File > Save** to save the changes.
7. Click the **Web Services Client Binding** tab. The Web Services Client Binding editor displays.
   a. Under Port Qualified Name Binding, select an existing entry or click **Add** to add a new port name binding. The Web Services Client Binding editor displays for the selected port.
   b. Under Login Binding, click **Edit** or **Enable**. This Login Binding dialog box displays.

1) In the **Authentication Method** field, enter the authentication method. The authentication method that you enter in this field must match the authentication method defined on the **Security Extension** tab for the same Web service port. This field is mandatory.
2) (Optional) Enter the token value type information in the **URI** and **Local name** fields. These fields are ignored for the BasicAuth, Signature, and IDAssertion authentication methods, but required for other authentication methods. The token value type information is inserted into the `<wsse:BinarySecurityToken>@ValueType` element for binary security token and is used as the namespace for the XML-based token.
3) Enter an implementation of the Java Authentication and Authorization Service (JAAS) `javax.security.auth.callback.CallbackHandler` interface. This is a mandatory field.
4) Enter the basic authentication information in the **User ID** and **Password** fields. The basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.
5) In the **Property** field, add name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as `java.util.Map`.
6) Click **OK**.

Click **Disable** under Login Binding on the **Web Services Client Port Binding** tab to remove the authentication method login binding.

c. Click **File > Save** to save the changes.

8. In the Package Explorer window, right-click the `webservices.xml` file and select **Open With > Web Services Editor**. The Web Services window displays.
   a. Click the **Security Extensions** tab. The Web Service Security Extensions editor displays.
      1) Under Web Service Description Extension, select an existing service reference or click **Add** to create a new extension.
      2) Under Port Component Binding, select an existing port qualified name of the selected service reference or click **Add** to create a new one.
      3) Under Request Receiver Service Configuration Details: Login Config, select an exiting authentication method or click **Add** and enter a new method in the Add AuthMethod field that displays. You can select multiple authentication methods for the request receiver. The security token of the incoming message is authenticated against the authentication methods in the order that they are specified in the list. Click **Remove** to remove the selected authentication method or methods.
   b. Click **File > Save** to save the changes.
   c. Click the **Bindings** tab. The Web Services Bindings editor displays.
      1) Under Web Service Description Bindings, select an existing entry or click **Add** to add a new Web services descriptor.
      2) Click the **Binding Configurations** tab. The Web Services Binding Configurations editor displays for the selected Web services descriptor.
      3) Under Request Receiver Binding Configuration Details: Login Mapping, click **Add** to create a new login mapping or click **Edit** to edit existing selected login mapping. The Login mapping dialog displays.
         a) In the **Authentication method** field, enter the authentication method. The information entered in this field must match the authentication

method defined on the **Security Extensions** tab for the same Web service port. This is a mandatory field.

b) In the **Configuration name** field, enter a JAAS login configuration name. You must define the JAAS login configuration name in the WebSphere Application Server Administrative Console under **Security > JAAS Configuration > Application Logins**). This is a mandatory field. For more information, see Configuring Java Authentication and Authorization Service login.

c) (Optional) Select **Use Token value type** and enter the token value type information in the **URI** and **Local name** fields. This information is optional for BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is used to validate the `<wsse:BinarySecurityToken>@ValueType` element for binary security tokens and to validate the namespace of the XML-based token.

d) Under Callback Handler Factory, enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the **Class name** field. This field is mandatory.

e) Under Callback Handler Factory Property, click **Add** and enter the name and value pairs for the Callback Handler Factory Property. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback .CallbackHandlerFactory.init()` method. The usage of these name and value pairs is determined by the CallbackHandlerFactory implementation chosen.

f) Under Login Mapping Property, click **Add** and enter the name and value pairs for the Login Mapping Property. These name and value pairs are available to the JAAS Login Module or Modules through the`com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback interface. Click **Remove** to delete selected login mapping.

g) Click **OK**.

d. Click **File > Save** to save the changes.

The previous steps define how to configure the request sender to create security tokens in the Simple Object Access Protocol (SOAP) message and the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

Once you have configured pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:

- Configuring the client for LTPA token authentication: specifying LTPA token authentication
- Configuring the client for LTPA token authentication: collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Configuring pluggable tokens using the administrative console

Prior to completing these steps, it is assumed that you have already created a Web services-enabled Java 2 Platform, Enterprise Edition (J2EE) with a Web Services for J2EE (JSR 109) enterprise application. If not, see Developing Web services to create Web services-enabled J2EE with a JSR 109 enterprise application. See either of the following topics for an introduction of how to manage Web services security binding information for the server:

- Configuring the server security bindings using the Assembly Toolkit
- Configuring the server security bindings using the administrative console

This document describes how to configure a pluggable token in the request sender (`ibm-webservicesclient-ext.xmi` and `ibm-webservicesclient-bnd.xmi` file) and request receiver (`ibm-webservices-ext.xmi` and `ibm-webservices-bnd.xmi` file).

Important: The pluggable token is required for the request sender and request receiver as they are a pair. The request sender and the request receiver must match for a request to be accepted by the receiver.

Complete the following steps to configure the client-side request sender (`ibm-webservicesclient-bnd.xmi` file) or server-side request receiver (`ibm-webservices-bnd.xmi` file) using the WebSphere Application Server Administrative Console.

1. Click **Applications > Enterprise Applications >** *enterprise_application*.
2. Under Related Items, click either **EJB Modules >** *Uri* or **Web Modules >** *Uri.* . The *Uri* is the Web services-enabled module
3. Under Additional Properties, click **Web Services: Client Security Bindings** to edit the response sender binding information, if Web services is acting as client.
4. Under Response Sender Binding, click **Edit**.
5. Under Additional Properties, click **Login Binding**.
6. Select **Dedicated Login Binding** to define a new login binding or select **None** to deselect the login binding.
   a. Enter the authentication method in the **Authentication Method** field. This entry must match the authentication method defined in IBM extension deployment descriptor. The authentication method must be unique in the binding file.
   b. Enter an implementation of the JAAS `javax.security.auth.callback.CallbackHandler` interface in the **Callback Handler**field.
   c. Optional: Enter the basic authentication user ID and password in the **Basic Auth User ID** and **Basic Auth Password** fields, respectively. The basic authentication information is passed to the construct of the CallbackHandler implementation. The usage of the basic authentication information is up to the implementation of the CallbackHandler.
   d. Enter the token value type Uniform Resource Identifier (URI) and local name in the **Token Type URI** and **Token Type Local Name** fields. This information is optional for the BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is inserted into the `<wsse:BinarySecurityToken>@ValueType` for binary security token and used as the namespace of the XML based token.
   e. Click **Apply**.

7. Under Additional Properties, click **Properties**. Define the property with name and value pairs. These pairs are passed to the construct of the CallbackHandler implementation as `java.util.Map`.
8. Click **Applications > Enterprise Applications >** *enterprise_application*.
9. Under Related Items, click either **EJB Modules >** *Uri* or **Web Modules >***Uri*.. The *Uri* is the Web services-enabled module
10. Under Additional Properties, click **Web Services: Server Security Bindings** to edit the request receiver binding information.
11. Under Request Receiver Binding, click **Edit**.
12. Under Additional Properties, click **Login Mappings**.
13. Click **New** to create new login mapping. You also can edit an existing login mapping by clicking on the name or delete a login mapping by selecting the box next to the login mapping name and clicking **Remove**.
    a. Enter the authentication method in the **Authentication Method** field. This entry must match the authentication method defined in the IBM extension deployment descriptor. The authentication method must be unique in the login mapping collection of the binding file.
    b. Select a JAAS Login Configuration name from the **JAAS Configuration Name** menu. The JAAS Login Configuration must be defined in **Security > JAAS Configuration > Application Logins**. For more information, see Configuring Java Authentication and Authorization Service login.
    c. Enter an implementation of the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory` interface in the **Callback Handler Factory Classname** field. This is a mandatory field.
    d. Enter the token value type Uniform Resource Identifier (URI) and local name in the **Token Type URI** and **Token Type Local Name** fields. This information is optional for the BasicAuth, Signature and IDAssertion authentication methods, but required for any other authentication method. The token value type is inserted into the `<wsse:BinarySecurityToken>@ValueType` for binary security token and used as the namespace of the XML based token.
    e. Click **Apply**.
14. Under Additional Properties, click **Properties**.
15. Click **New** and enter the name and value pairs in the **Property Name** and **Property Value** fields. These name and value pairs are available to the JAAS Login Module or Modules by `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback` JAAS Callback. These pairs are available when editing existing login mappings, but not when creating new login mappings.
16. Return to the Additional Properties heading and click **Callback Handler Factory Property**. The **Callback Handler Factory Property** option is located on the same menu where you previously clicked **Properties**.
17. Click **New** and enter the name and value pairs in the **Property Name** and **Property Value** fields. These name and value pairs are passed as `java.util.Map` to the `com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory.init()` method. The usage of these name and value pairs is up to the CallbackHandlerFactory implementation.
18. Click **Save** in the upper-left section of the administrative console.

The previous steps define how to configure the request sender to create security tokens in the Simple Object Access Protocol (SOAP) message and the request receiver to validate the security tokens found in the incoming SOAP message. WebSphere Application Server supports pluggable security tokens.

You can use the authentication method defined in the login bindings and login mappings to generate security tokens in the request sender and validate security tokens in the request receiver.

Once you have configured pluggable tokens, you must configure both the client and the server to support pluggable tokens. See the following topics to configure the client and the server:

- Configuring the client for LTPA token authentication: specifying LTPA token authentication
- Configuring the client for LTPA token authentication: collecting the authentication information
- Configuring the server to handle LTPA token authentication
- Configuring the server to validate LTPA token authentication information

Pluggable token support

You can extend the WebSphere Application Server login mapping mechanism to handle new types of authentication tokens.WebSphere Application Server provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). Pluggable security token support provides plug-in points to support customer security token types including token generation, token validation, and mapping a client identity to a WebSphere Application Server identity that is used by the Java 2, Enterprise Edition (J2EE) authorization engine. Moreover, the pluggable token generation and validation framework allows XML-based tokens to be inserted into the Web service message header and validated on the receiver side.
You can extend the WebSphere Application Server login mapping mechanism to handle new types of authentication tokens.WebSphere Application Server provides a pluggable framework to generate security tokens on the sender-side of the message and to validate the security token on the receiver-side of the message. The framework is based on the Java Authentication and Authorization Service (JAAS) Application Programming Interfaces (APIs). Pluggable security token support provides plug-in points to support customer security token types including token generation, token validation, and mapping a client identity to a WebSphere Application Server identity that is used by the Java 2, Enterprise Edition (J2EE) authorization engine. Moreover, the pluggable token generation and validation framework allows XML-based tokens to be inserted into the Web service message header and validated on the receiver side.

Users can use the javax.security.auth.callback.CallbackHandler implementation to create a new type of security token following these guidelines:

- Use a constructor that takes a user name (a string or null, if not defined), password (a char[] or null, if not defined) and `java.util.Map` (empty, if properties are not defined).
- Use handle() methods that can process the `javax.security.auth.callback.NameCallback`, `javax.security.auth.callback.PasswordCallback`, `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback`, and `com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl`

implementations. If either thejavax.security.auth.callback.NameCallback or the javax.security.auth.callback.PasswordCallback implementation is populated with data, then a <wsse:UsernameToken> element is created. Otherwise, if om.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl is populated, the <wsse:BinarySecurityToken> element is created from the com.ibm.websphere.security.auth.callback.WSCredTokenCallbackImpl implementation. Lastly, if com.ibm.wsspi.wssecurity.auth.callback.XMLTokenCallback is populated, a XML-based token is created based on the Document Object Model (DOM) element that is returned from the XMLTokenCallback. Encode the token byte by using the security handler and not by using the javax.security.auth.callback.CallbackHandler implementation.

You can implement the com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory interface, which is a factory for instantiating the javax.security.auth.callback.CallbackHandler. For your own implementation, you must provide the javax.security.auth.callback.CallbackHandler interface. The Web service security run time instantiates the factory implementation class and passes the authentication information from the Web services message header to the factory class through the setter methods. The Web services security run time then invokes the newCallbackHandler() method of the factory implementation class to obtain an instance of the javax.security.auth.CallbackHandler object. The object is passed to the JAAS login configuration.

The following is the definition of the CallbackHandlerFactory interface:

```
public interface com.ibm.wsspi.wssecurity.auth.callback.CallbackHandlerFactory {
       public void setUsername(String username);
       public void setRealm(String realm);
       public void setPassword(String password);
       public void setHashMap(Map properties);
       public void setTokenByte(byte[] token);
       public void setXMLToken(Element xmlToken);
       public CallbackHandler newCallbackHandler();
```

Configuring the client for LTPA token authentication: specifying LTPA token authentication

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a Web service calls a downstream Web service, you can configure the first Web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a Web service acting as a client to a downstream Web service. In order for the downstream Web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify LTPA token as the authentication method:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.

4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Sender Configuration > Login Config** section.
8. Select **LTPA** as the authentication method. For more conceptual information on LTPA authentication, see LTPA.

Once you have specified LTPA token as the authentication method, you must specify how to collect the LTPA token information. See Configuring the client for LTPA token authentication: collecting the authentication information for more information.

Configuring the client for LTPA token authentication: Collecting the authentication method information

Use this task to configure Lightweight Third-Party Authentication (LTPA) token authentication. Do not configure the client for LTPA token authentication unless the authentication mechanism configured in WebSphere Application Server is LTPA. When a client authenticates to a WebSphere Application Server, the credential created contains an LTPA token. When a Web service calls a downstream Web service, you can configure the first Web service to send the LTPA token from the originating client. Do not attempt to configure LTPA from a pure client. LTPA works only when you configure the client-side of a Web service acting as a client to a downstream Web service. In order for the downstream Web service to validate the LTPA token, the LTPA keys on both servers must be the same.

Complete the following steps to specify how to collect the LTPA token authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservicesclient.xml` file, select **Open With > Web Services Client Editor**.
6. Click the **Port Binding** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Security Request Sender Binding Configuration > Login Binding** section.
8. Click **Edit** to view the login binding information and select **LTPA**. If LTPA is not already there, enter it as an option. The login binding dialog displays and either select or enter the following information:

   Authentication method
   : The authentication method specifies the type of authentication that occurs. Select **LTPA** to use identity assertion.

   Token value type URI and token value type local name
   : When you select **LTPA**, you must edit the **token value type URI** and the **local name** fields. These values are specifically for custom

authentication types, which are authentication methods not mentioned in the specification. For the **token value type URI** field, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For the **local name** field, enter the following string: `LTPA`.

Callback handler
The callback handler specifies the Java Authentication and Authorization Service (JAAS) callback handler implementation for collecting the LTPA information. Specify the `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler` implementation for LTPA.

Basic authentication user ID and basic authentication password
For LTPA, you can leave these fields empty.

Property name and property value
For LTPA, you can leave these fields empty.

See Configuring the client for LTPA token authentication: specifying LTPA token authentication if you have not previously specified this information.

Configuring the server to handle LTPA token authentication information

This task is used to configure Lightweight Third-Party Authentication (LTPA). LTPA is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first Web service, which authenticated the originating client, to the downstream Web service. Do not attempt to configure LTPA from a pure client. Once the downstream Web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify that LTPA is authentication method. The authentication method indicated in these steps must match the authentication method specified for the client.

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Security Extensions** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Service Configuration Details > Login Configuration** section. You can select from the following options:
   - BasicAuth
   - Signature
   - ID assertion
   - LTPA
8. Select **LTPA** to authenticate the client using the LTPA token received from the request.

Once you have specified the authentication method, you must specify the information that the server must validate. See Configuring the server to validate LTPA token authentication information for more information.

Configuring the server to validate LTPA token authentication information

This task is used to configure Lightweight Third-Party Authentication (LTPA). LTPA is a type of authentication mechanism in WebSphere Application Server security that defines a particular token format. The purpose of the LTPA token authentication is to flow the LTPA token from the first Web service, which authenticated the originating client, to the downstream Web service. Do not attempt to configure LTPA from a pure client. Once the downstream Web service receives the LTPA token, it validates the token to verify that the token has not been modified and has not expired. For validation to be successful, the LTPA keys used by both the sending and receiving servers must be the same.

Complete the following steps to specify how the server must validate the LTPA token authentication information:

1. Launch the Assembly Toolkit.
2. Open the J2EE perspective by clicking **Window > Open Perspective > Other > J2EE**.
3. Select the Web services enabled EJB or Web module.
4. In the Project Navigator window, locate the **META-INF** directory for an EJB module or the **WEB-INF** directory for a Web module.
5. Right-click the `webservices.xml` file, select **Open With > Web Services Editor**.
6. Click the **Binding Configurations** tab, which is located at the bottom of the Web Services Client Editor within the Assembly Toolkit.
7. Expand the **Request Receiver Binding Configuration Details > Login Mapping** section.
8. Click **Edit** to view the Login Mapping information. The login mapping information displays and either select or enter the following information:

   Authentication method
   The authentication method specifies the type of authentication that occurs. Select **LTPA** to use LTPA token authentication.

   Configuration name
   This name specifies the Java Authentication and Authorization Service (JAAS) login configuration name. For the LTPA authentication method, enter `WSLogin` for the JAAS login configuration name. This configuration understands how to validate an LTPA token.

   Use token value type
   This option determines if you want to specify a custom token type. For LTPA authentication, you must select this option because LTPA is considered a custom type. LTPA is not in the Web Services Security Specification.

   Token value type URI and local name
   If you select **Use Token value type** you must enter data into the **Token value Type URI** and **local name** fields. For URI, enter the following string: `http://www.ibm.com/websphere/appserver/tokentype/5.0.2`. For local name, enter the following string: `LTPA`

Callback handler factory class name
: This classname creates a JAAS CallbackHandler implementation that understands the following callback handlers:
  - `javax.security.auth.callback.NameCallback`
  - `javax.security.auth.callback.PasswordCallback`
  - `com.ibm.wsspi.wssecurity.auth.callback.BinaryTokenCallback`
  - `com.ibm.wsspi.wssecurity.auth.callback.XMLTokenReceiverCallback`
  - `com.ibm.wsspi.wssecurity.auth.callback.PropertyCallback`

  For any of the default Authentication methods (BasicAuth, IDAssertion, Signature, LTPA), use the callback handler factory default implementation. Enter the following class name for any of the default authentication methods including LTPA: `com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`

  This implementation creates the correct callback handler for the default implementations.

Callback handler factory property
: This field is used to specify callback handler properties for custom callback handler factory implementations. The default callback handler factory implementation does not need you to specify any properties. For LTPA, you do not need to enter any properties for this field.

Login mapping property
: This field is used to specify properties for a custom login mapping. For the default implementations including LTPA, you do not need to enter any properties for this field.

See Configuring the server to handle LTPA token authentication if you have not previously specified this information.

Lightweight Third Party Authentication:

When you use the lightweight third party authentication (LTPA) method, the security token generated is `<wsse:BinarySecurityToken>`. On the request sender side, the security token is generated by invoking a callback handler. On the request receiver side, the security token is validated by a Java Authentication and Authorization Service (JAAS) login module.

The token generation and token validation operations are described below.

LTPA token generation
: The request sender uses a callback handler to generate an LTPA security token. The callback handler returns a security token that is inserted in the SOAP message. Specify the appropriate callback handler in the `<LoginBinding>` element of the bindings file (`ibm-webservicesclient-bnd.xmi`). The following is a callback handler implementation that can be used with the LTPA authentication method:

  `com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler`

  You can add your own callback handlers that implement `javax.security.auth.callback.CallbackHandler`.

  When using the LTPA authentication method (or any authentication method other than BasicAuth, Signature or IDAssertion), the

TokenValueType attribute of the `<LoginBinding>` element in the bindings file (`ibm-webservicesclient-bnd.xmi`) must be specified. The values to use for the LTPA TokenValueType are:

```
uri="http://www.ibm.com/websphere/appserver/tokentype/5.0.2"
localName="LTPA"
```

LTPA token validation

The request receiver retrieves the LTPA security token from the SOAP message and validates it using a JAAS login module. The security token, `<wsse:BinarySecurityToken>`, is used to perform the validation. If the validation is successful, the login module returns a JAAS Subject. Subsequently, this Subject is set as the identity of the thread of execution. If the validation fails, the request is rejected with a SOAP fault.

The appropriate JAAS login configuration to use is specified in the bindings file `<LoginMapping>` element. There are default bindings specified in the `ws-security.xml` file, but these can be overridden using the application-specific `ibm-webservices-bnd.xmi` file. The configuration information consists of a CallbackHandlerFactory, ConfigName and TokenValueType. The CallbackHandlerFactory specifies the name of a class to use to create the JAAS CallbackHandler object. A CallbackHandlerFactory implementation is provided (`com.ibm.wsspi.wssecurity.auth.callback.WSCallbackHandlerFactoryImpl`). The ConfigName specifies a JAAS configuration name entry. The Web services security run time first searches the `security.xml` file for a matching entry and if a matching entry is not found, the run time searches the `wsjaas.conf` file. A default configuration entry suitable for the LTPA authentication method is provided (`WSLogin`). There is an appropriate TokenValueType element in the LTPA LoginMapping section of the default `ws-security.xml` file.

Tuning Web services based on Web Services for J2EE

Performance considerations are the same for Web services applications that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) and regular J2EE applications. See Tuning performance for more information about analyzing and tuning J2EE applications.

You can use the Performance Monitoring Infrastructure (PMI) to measure the time required to process Web services requests. To monitor Web services application performance:

1. Enable PMI services in application server through the administrative console. Select the Web Service module, named webServicesModule, in step 7.
2. Monitor performance with Tivoli Performance Viewer In the left-hand pane of the performance view, expand the host and server and select **Web Services**. Run the Web services client application.

Measurements are available for the following items:

- Number of Web services loaded by the application server
- Number of requests received
- Number of requests dispatched to an implementation bean
- Number of requests dispatched with successful replies
- Average time in milliseconds between receiving the request and returning the reply
- Average time in milliseconds between receiving the request and dispatching it to the bean

- Average time in milliseconds between dispatch and receipt of reply from the bean
- Average time in milliseconds between receipt of reply from bean to return of result to client
- Average size of request and reply
- Average size of request
- Average size of reply

Troubleshooting Web services based on Web Services for J2EE

Select the Web services topic area you want to troubleshoot:

- Command-line tools
- Java compiler errors
- Runtime errors and exceptions
- Client runtime errors and exceptions
- Serialization or deserialization errors

Troubleshooting command-line tools for Web services based on Web Services for J2EE

This topic discusses troubleshooting command-line tools for Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

WSDL2Java command-line tool

Emitter failure error occurs when running the WSDL2Java command on a WSDL document containing a JMS-style endpoint URL

If you run the **WSDL2Java** command-line tool on a WSDL document that contains a JMS-style endpoint URL , for example `jms:/...`, the `urlprotocols.jar` file that contains the custom protocol handler for the JMS protocol must be in the CLASSPATH. The error **`WSWS3099E: Error: Emitter failure. Invalid endpoint address in port <x> in service <y>: <jms-url-string>`** can be avoided by making sure the `urlprotocols.jar` file is in the CLASSPATH.

To add the `urlprotocols.jar` file to the CLASSPATH:

On Windows platforms, edit the *install_root*`\bin\setupCmdLine.bat` and locate the line which sets the *WAS_CLASSPATH* environment variable. Add `%`*install_root*`%\lib\urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

On UNIX platforms, edit the *install_root*`/bin/setupCmdLine.sh` file and add `$`*install_root*`/lib/urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

Make sure to use the proper deliminator character for your platform, for example, use a semi-colon (;) for Windows platforms and a colon (:) for UNIX platforms.

Troubleshooting compiled bindings for Web services based on Web Services for J2EE

This topic discusses troubleshooting compiled bindings of Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

Context root not recognized when mapping the default XML namespace to a Java package

When you map the default XML namespace to a Java package the context root is not recognized. If two namespaces are the same up to the first slash, they are mapped to the same Java package. For example, the XML namespaces `http://www.ibm.com/foo` and `http://www.ibm.com/bar` both map to the Java package `www.ibm.com`. Use the -NStoPkg option of the **Java2WSDL** command to specify the package for the fully qualified namespace.

Java code to WSDL mapping cannot be reversed back to the original Java code

If you find that a WSDL file you created with the **Java2WSDL** command-line tool cannot be compiled when regenerated into Java code using the **WSDL2Java** command-line tool it is because the Java API for XML-based remote procedure call (JAX-RPC) mapping from Java code to WSDL is not reversible back to the original Java code.

To troubleshoot this problem review the WSDL file that was generated by the **Java2WSDL** tool using the information in Mapping between Java, WSDL and XML and the JAX-RPC specification available through Web services: Resources for learning. Use this information to determine which elements in the WSDL file are causing the problem. You can modify the WSDL file, or the original Java interface used to generate the WSDL file, and run the **Java2WSDL** command again.

Troubleshooting the run time of Web services based on Web Services for J2EE

This topic discusses troubleshooting the run time of Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can troubleshoot run time errors and exceptions as follows:
- Trace SOAP messages
- Trace the components of Web services based Web Services for J2EE

Tracing SOAP messages

This topic discusses tracing SOAP messages that request Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

You can trace the SOAP messages exchanged between a client and the server using the **TCPMon** command tool. The **TCPMon** command redirects messages from one port to another and records them. The WebSphere Application Server listens on port 9080. To trace messages sent to the application server, the **TCPMon** command is configured to listen on port 9088 and redirect them to 9080. The client is redirected to use port 9088 to access the Web service.

Redirecting an application client to a different port is most easily done by changing the SOAP address in the client's Web Services Description Language (WSDL) file to use port 9088 and then running the **wsdeploy** command-line tool on the client enterprise archive (EAR) file to regenerate the service implementation.

You should confirm that the server providing the Web service is running. The following task is performed on the machine providing the Web service.

To trace SOAP messages in Web services:

1. Set up a development and unmanaged client execution environment for Web services based on Web Services for J2EE
2. Run the **java -Djava.ext.dirs=%WAS_EXT_DIRS%** command. A window labeled TCPMonitor displays.
3. Configure the TCPMonitor to listen on port 9088 and forward messages to port 9080.
   a. In the **Listen Port #** field, enter `9088`.
   b. Click **Listener**
   c. In the **TargetHostname** field, enter `localhost`.
   d. In the **Target Port #** field, enter `9080`.
   e. Click **Add**.
   f. Click on the **Port 9088** tab that displays on the top of the page.

The messages exchanged between the client and server appear in the TCPMonitor Request and Response pane.

Save the message data and analyze it.

Tracing Web services components based on Web Services for J2EE

The following are tasks in which you can enable trace for Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

1. Enable trace for a Web services unmanaged client.
   a. Create a trace properties file by copying `%install_root%\WebSphere\AppServer\properties\TraceSettings.properties` file to the same directory as your client application Java archive (JAR) file.
   b. Edit the properties file and change the value of `traceFileName` to output the trace data. For example, `traceFileName=c:\\temp\\myAppClient.trc`.
   c. Edit the properties file to remove `com.ibm.ejs.ras.*=all=enabled` and add `com.ibm.ws.webservices.*=all=enabled`.
   d. Add the option `-DtraceSettingsFile=<trace_properties_file>` to the Java command- line used to run the client, where *trace_properties_file* represents the name of the properties file created in steps 1-2. For example, **java -DtraceSettingsFile=TraceSettings.properties myApp.myAppMainClass**.
2. Enable trace for a Web services managed client.
   a. Invoke the **launchClient** command-line tool with the following options: -CCtrace=com.ibm.ws.webservices.*=all=enabled-CCtracefile=traceFileName For example:

      ***%install_root%*\bin\launchClient MyAppClient.ear-CCtrace=com.ibm.ws.webservices.*=all=enabled -CCtracefile=myAppClient.trc**

See launchClient tool for more information.

3. Enable trace for a Web Services for J2EE server application.
   a. Start WebSphere Application Server.
   b. Open the administrative console.
   a. Click **Servers** >**Application Servers** > *server*.
   a. Click **Diagnostic Trace Service**.
   a. In the **Trace Specification** field, delete the text ***=all=enabled** and add **com.ibm.ws.webservices.*=all=enabled**.
   b. Click **Save** and **Apply**.

Troubleshooting the run time for a Web services client based on Web Services for J2EE

This topic discusses troubleshooting Web services clients that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

Malformed URL exception displays when running a client that uses a JMS-style endpoint URL

If you are using the **launchClient** command to run a managed or unmanaged client that uses a JMS-style endpoint URL, the `urlprotocols.jar` file that contains the custom protocol handler for the JMS protocol must be in the CLASSPATH. The malformed URL exception can be avoided by making sure the `urlprotocols.jar` file is in the CLASSPATH.

To add the `urlprotocols.jar` file to the CLASSPATH:

On Windows platforms, edit the *install_root*`\bin\setupCmdLine.bat` and locate the line which sets the *WAS_CLASSPATH* environment variable. Add `%`*install_root*`%\lib\urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

On UNIX platforms, edit the *install_root*`/bin/setupCmdLine.sh` file and add `$`*install_root*`/lib/urlprotocols.jar` to the end of the line that sets the *WAS_CLASSPATH* environment variable.

Troubleshooting serialization and deserializaton in Web services based on Web Services for J2EE

The following are problems you might encounter performing serialization and deserialization in Web services that are developed and implemented based on the Web Services for Java 2 platform Enterprise Edition (J2EE) specification.

Time zone information in deserialized java.util.Calendar is not as expected

When the client and server are based on Java code and a `java.util.Calendar` is received, the time zone in the received `java.util.Calendar` instance might be different from that of the `java.util.Calendar` instance that was sent.

This occurs because `java.util.Calendar` is encoded as an xsd:dateTime for transmission. An xsd:dateTime is required to encode the correct time (base time plus or minus a time zone offset), but is not required to preserve locale information, including the original time zone.

The fact that the time zone for the current locale is not preserved needs to be accounted for when comparing Calendar instances. The `java.util.Calendar` class equals method checks that the time zones are the same when determining equality. Since the time zone in a deserialized Calendar instance might not match the current locale, the before and after comparison methods should be used to test that two Calendars refer to the same date and time as shown below:

```
java.util.Calendar c1 = ...// Date and time in time zone 1
java.util.Calendar c2 = ...// Same date and equivalent time, but in time zone 2

// c1 and c2 are not equal because their time zones are different
if (c1.equals (c2)) System.out.println("c1 and c2 are equal");

// but c1 and c2 do compare as "not before and not after" since they represent
the same date and time
if (!c1.after(c2) & !c1.before(c2) {
   System.out.println("c1 and c2 are equivalent");
}
```

Mixing Web services client and server bindings causes exceptions

Web Services for J2EE and Java API for XML-based remote procedure call (JAX-RPC) do not support ″round-trip″ mapping between Java code and a Web Services Description Language (WSDL) document for all Java types. For example, you cannot turn (serialize) a Java Date into XML code and then turn it back (deserialize) into a Java Date. It deserializes as Java Calendar.

If you have a Java implementation that you create a WSDL document from, and you generate client bindings from the WSDL document, the client classes can be different from the server classes even though the client classes have the same package and class names. The Web service client classes must be kept separate from the Web service server classes. For example, do not place the Web service server bindings classes in a utility Java archive (JAR) file and then include a Web service client JAR file that references the same utility JAR file.

If you do not keep the Web service client and server classes separate, a variety of exceptions can occur, depending on the Java classes used. The following is a sample stack trace error that can occur:

```
com.ibm.ws.webservices.engine.PivotHandlerWrapper TRAS0014I:
The following exception was loggedjava.lang.NoSuchMethodError:
 com.ibm.wssvt.acme.websvcs.ExtWSPolicyData:
 method getStartDate()Ljava/util/Date;
not found at
 com.ibm.wssvt.acme.websvcs.ExtWSPolicyData_Ser.addElements(ExtWSPolicyData_Ser.java: 210)
at com.ibm.wssvt.acme.websvcs.ExtWSPolicyData_Ser.serialize (ExtWSPolicyData_Swer.java:29)
at com.ibm.ws.webservices.engine.encoding.SerializationContextImpl.serializeActual
  (SerializationContextImpl.java 719)
at com.ibm.ws.webservices.engine.encoding.SerializationContextImpl.serialize
  (SerializationContextImpl.java: 463)
```

The problem is caused by using an interface like the following for the Service Endpoint Interface in the service implementation:

```
package server:
public interface Test_SEI extends java.rmi.Remote {
 public java.util.Calendar getCalendar () throws java.rmi.RemoteException;
 public java.util.Date getDate() throws java.rmi.RemoteException;
}
```

When this interface is compiled and run through the **Java2WSDL** command-line tool, the WSDL document maps the methods as follows:

```
<wsdl:message name="getDateResponse">
 <wsdl:part name="getDateReturn" type="xsd:dateTime"/>
</wsdl:message>

<wsdl:message name="getCalendarResponse">
 <wsdl:part name="getCalendarReturn" type="xsd:dateTime"/>
</wsdl:message>
```

The JAX-RPC mapping implemented by the **Java2WSDL** tool has mapped both java.util.Date and java.util.Calendar to the XML type xsd:dateTime. The next step is to use the generated WSDL file to create a client for the Web service. When you run the **WSDL2Java** command-line tool on the generated WSDL, the generated classes include a different version of server.Test_SEI, for example:

```
package server;
public interface Test_SEI extends java.rmi.Remote {
 public java.util.Calendar getCalendar() throws java.rmi.RemoteException;
 public java.util.Date getDate() throws java.rmi.RemoteException;
}
```

Note: The client version of the `service.Test_SEI` interface is different from the server version in that both `getCalendar` and `getDate` methods return `java.util.Calendar`. The serialization and deserilazation code that the client expects is the client version of the SEI. If the server version inadvertently appears in the client's CLASSPATH, at either compilation or execution time, an exception occurs.

In addition to the`NoSuchMethod` error, the `IncompatibleClassChangeError` and `ClassCastException` can occur, however, almost any run-time exception can occur. The best practice is to be diligent about separating client Web services bindings classes from server Web services bindings classes. The client bindings classes and server bindings classes should never be placed in the same module and, if they are in the same application, should not have bindings classes in utility JAR files that are shared between modules.

Frequently asked questions about Web services based on Web Services for J2EE

This topic presents frequently asked questions about Web services that are developed and implemented based on the Web Services for Java 2 platform, Enterprise Edition (J2EE) specification.

- What IBM development tools work with Web Services for J2EE?
- Is Web Services for J2EE part of the J2EE specification?
- What is the relationship between Web Services for J2EE and the Web Service Invocation Framework (WSIF)?
- What is the relationship between Apache SOAP 2.3 and Web Services for J2EE?
- What is the relationship between the Apache Axis component of the Web services technology preview available with WebSphere Application Server 5.0 and Web Services for J2EE?
- What standards does the Web Services for J2EE component of WebSphere Application Server 5.0. support?
- Does Web Services for J2EE interoperate with other SOAP implementations, like .NET?
- Why can I not use a JavaBean to implement a SOAP Java Messaging Service (JMS) service?
- Does the SOAP JMS support interoperate with other vendors?
- How does two-way messaging with SOAP JMS work? Can it support multiple clients making simultaneous requests?

What IBM development tools work with Web Services for J2EE?

WebSphere Studio Application Developer Version 5.1 and the Assembly Toolkit Version 5.1 both support the use of Web Services for J2EE. The Application Assembly Tool, included with Websphere Application Server, and Websphere Studio Application Developer versions earlier than Version 5.01, do not support Web Services for J2EE.

Is Web Services for J2EE part of the J2EE specification?

For WebSphere Application Server 5.0.2, the Web Services for J2EE Version 1.0 specification is an addition to J2EE 1.3. J2EE 1.4 requires support for Web Services for J2EE Version 1.1. There are minor differences between the J2EE 1.3 Version (JSR-109 Version 1.0) and the J2EE 1.4 Version (JSR-109 Version 1.1).

What is the relationship between Web Services for J2EE and the Web Service Invocation Framework (WSIF)?

Web Services for J2EE and WSIF represent two different programming models for accessing Web services. Web Services for J2EE is standard, Java-centric, and more statically bound to WSDL documents due to the use of generated stubs. WSIF directly models Web Services Description Language (WSDL) documents. WSIF is more suitable when dynamically interpreting WSDL. Future versions of WebSphere Application server will leverage both technologies to achieve dynamic, high performing standards-based Web services implementations.

What is the relationship between Apache SOAP 2.3 and Web Services for J2EE?

Apache SOAP shipped with WebSphere Application Server Versions 4.0 and 5.0. It continues to co-exist with Web Services for J2EE. Apache SOAP is a proprietary API and applications written for it are not portable to other SOAP implementations. Applications written for Web Services for J2EE should be portable to any vendor's implementation that supports Web Services for J2EE.

What is the relationship between the Apache Axis component of the Web services technology preview available with WebSphere Application Server 5.0 and Web Services for J2EE?

The Web services technology preview leveraged the work that IBM contributed to the Apache Axis code base. The Web Services for J2EE support included with WebSphere Application Server 5.0.2 is derived from Apache Axis, but has diverged and contains many IBM-specific features to enhance performance, scalability, reliability, interoperability, and integration with the WebSphere Application Server.

What standards does the Web Services for J2EE component of WebSphere Application Server 5.0. support?

The following standards are supported by the Web Services for J2EE component of WebSphere Application Server 5.0:
- SOAP Version 1.1
- Web Services Description Language (WSDL) Version 1.1
- Web Services for J2EE (JSR-109) Version 1.0
- Java API for XML-Based RPC (JAX-RPC) Version 1.0
- SOAP with attachments API for Java (SAAJ) Version 1.1

Does Web Services for J2EE interoperate with other SOAP implementations, like .NET?

WebSphere Application Server Version 5.0.2 and Version 5.1 support Web services that are consistent with the the WS-I Basic Profile 1.0, and should interoperate with any other vendor conforming to this specification.

Why can I not use a Java bean to implement a SOAP Java Messaging Service (JMS) service?

The SOAP JMS support uses Message Driven Beans (MDB) to implement the JMS endpoint. MDBs can only be used in the EJB container and delegate to an enterprise bean. If you want to use a Java bean instead of an enterprise bean to implement the service endpoint, you must create a ″facade″ enterprise bean that delegates to the Java bean.

Does the SOAP JMS support interoperate with other vendors?

No. There is currently no specification for SOAP JMS, therefore each vendor chooses its own implementation technique.

How does two-way messaging with SOAP JMS work? Can it support multiple clients making simultaneous requests?

Before a client issues a two-way request, it creates a temporary JMS queue to receive the response. This temporary queue is specified as the **replyTo** destination in the outgoing JMS request message. After the server processes the request, it directs the response to the **replyTo** destination specified in the request message. The client deletes the temporary queue after the response has been received. The server is able to handle simultaneous requests from multiple clients since each incoming request message contains the destination to which the reply should be sent.

Web services: Resources for learning

This topic provides relevant supplemental information about the following Web services-related topics:

- Web services overview

  Including the WebSphere Version 5 Web Services Handbook
- Developing Web services:

  Including developing Web services based on the Java 2 platform, Enterprise Edition (J2EE) and Java API for XML-based remote procedure call (JAX-RPC) specifications.
- Universal Description Discovery and Integration (UDDI)

  Including an overview about UDDI and information about the UDDI Java API.
- Web Services Invocation Framework (WSIF)

  A look into the Apache Software Foundation and its maintenance of WSIF.
- SOAP

  Including an overview about SOAP and information about the SOAP syntax and processing rules.
- Security

  Including a roadmap to security, the WS-Security specification, best practices, a profile of the OASIS Security Assertion Markup Language (SAML) and more.
- Samples

Includes WebSphere Application Server Samples Gallery and Samples Central for Web services gateway, UDDI and WSIF.
- Other references

The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Web services overview

- WebSphere Version 5 Web Services Handbook

  This IBM Redbook describes the new concept of Web services from various perspectives. It presents the major building blocks Web services rely on. Well-defined standards and new concepts are presented and discussed.
- IBM Web Services architecture debuts

  Introducing IBM Web services, a distributed software architecture of service components. This brief overview and in-depth interview on IBM DeveloperWorks cover the fundamental concepts of Web services architecture and what they mean for developers. The interview with IBM professional Rod Smith explores which types of developers Web services targets, how Web services reduces development time, what developers could be doing with Web services now, and takes a glance at the economics of dynamically discoverable services.
- Web services (r)evolution, Part 1

  This article focuses on the benefits and challenges of building Web services applications. Web services might be an evolutionary step in designing distributed applications, however, they are not without their problems. Outlined are the difficulties developers face in creating a truly workable distributed system of Web services. This article also outlines author Graham Glass' plan for building peer-to-peer Web applications.

Developing Web services

- JSR 109: Implementing Enterprise Web services

  This document describes the Java 2 platform, Enterprise Edition (J2EE) specification.
- Java API for XML-based RPC (JAX-RPC): Core Web services API in the Java platform

  This document reviews the JAX-RPC specification which enables Java technology developers to develop SOAP-based interoperable and portable Web services.
- Web Services Description Language

  This article is a detailed overview of Web Services Description Language (WSDL), which includes programming specifications.

UDDI

- Universal Description, Discovery and Integration

  This article is a detailed overview of Universal Description, Discovery and Integration (UDDI).
- UDDI4J: Matchmaking for Web services

Reviewed in this article are the basics of UDDI, the Java API to UDDI, and how you can use this technology to start building, testing, and deploying your own Web services.

WSIF

- The Apache Software Foundation. The Apache Software Foundation provides support for the Apache community of open-source software projects. Of particular interest is the Apache Web services project. The WSIF source code has been donated by IBM to the Apache Software Foundation, and is maintained here as an Apache project.

SOAP

- SOAP

  This article is a detailed overview of SOAP, which includes programming specifications.
- SOAP Security Extensions: Digital Signature

  This document specifies the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 Envelope

Security

- Security in a Web Services World: A Proposed Architecture and Roadmap

  This document describes a proposed model for addressing security within a Web service environment. It defines a comprehensive Web Services Security model that supports, integrates, and unifies several popular security models, mechanisms, and technologies, including both symmetric and public key technologies, in a way that enables a variety of systems to securely interoperate in a platform and language-neutral manner. It also describes a set of specifications and scenarios that show how these specifications can be used together.
- Web Services Security (WS-Security)

  The Web Services Security specifications describe enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies. Web Services Security also provides a general-purpose mechanism for associating security tokens with messages. Additionally, Web Services Security describes how to encode binary security tokens. Specifically, the specification describes how to encode X.509 certificates and Kerberos tickets, as well as how to include opaque encrypted keys. It also includes extensibility mechanisms that can be used to further describe the characteristics of the credentials that are included with a message.
- SOAP Security Extensions: Digital Signature

  This document specifies the syntax and processing rules of a SOAP header entry to carry digital signature information within a SOAP 1.1 Envelope
- Web Services Security Addendum

  This document describes clarifications, enhancements, best practices, and errata of the Web Services Security specification.
- WS-Security Profile of the OASIS Security Assertion Markup Language (SAML) Working Draft 04, 10 September 2002

  This document proposes a set of standards for SOAP extensions used to increase message confidentiality.
- Web Services Security: SOAP Message Security Working Draft 12, Monday 21 April 2003

This document describes the support for multiple token formats, trust domains, signature formats, and encyrption technologies.

- JSR 55:Certification Path API

  This document provides a short description of the certification path API.
- XML-Signature Syntax and Processing

  This document specifies XML digital signature processing rules and syntax. XML signatures provide integrity, message authentication, or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.
- Canonical XML Version 1.0

  This specification describes a method for generating a physical representation, the canonical form, of an XML document that accounts for the permissible changes.
- Exclusive XML Canonicalization Version 1.0

  Canonical XML [XML-C14N] specifies a standard serialization of XML that, when applied to a subdocument, includes the subdocument's ancestor context including all of the namespace declarations and attributes in the ″xml:″namespace.
- XML Encryption Syntax and Processing

  This document specifies a process for encrypting data and representing the result in XML.
- Decryption Transform for XML Signature

  This document specifies an XML Signature ″decryption transform″ that enables XML Signature applications to distinguish between those XML Encryption structures that were encrypted before signing, and must not be decrypted, and those that were encrypted after signing, and must be decrypted, for the signature to validate.
- WS-Security

  This document specifies resources for the April 2002 Web Services Security Specification. The following addenda and drafts are available:
  - http://schemas.xmlsoap.org/ws/2002/07/secext/
  - http://schemas.xmlsoap.org/ws/2002/07/utility/
  - OASIS draft 12 for secext
  - OASIS draft 12 for utility
- XML Encryption Syntax and Processing W3C Recommendation 10 December 2002
- XML-Signature Syntax and Processing W3C Recommendation 12 February 2002
- Web Services Security Addendum
- Web Services Security Core Specification Working Draft 01, 20 September 2002
- Web Services Security: SOAP Message Security Working Draft 13, Thursday, 01 May 2003
- Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, RFC3280, April 2002
- OASIS Web Services Security Technical Committee

Samples

- Samples Gallery
- Samples Central. Samples and associated documentation for the following Web services components are available through the Samples Central page of the IBM WebSphere Developer Domain Web site:
  - The IBM private UDDI registry.
  - The Web Services Invocation Framework (WSIF).

Other references

- The Apache Software Foundation. The Apache Software Foundation provides support for the Apache community of open-source software projects. Of particular interest is the Apache Web services project.
- Web services insider, Part 1: Reflections on SOAP

  What is the current state of the *Web services revolution*? Find out at this Web site that features the column *Web services insider, Part 1*. The author answers this question by reviewing the tools and technologies that have emerged over the past year, highlighting their differences and similarities.
- The Web services insider, Part 2: A summary of the W3C Web Services Workshop

  This is a brief summary of a W3C Web services workshop.

Chapter 8. Web Services Invocation Framework (WSIF): Enabling Web services

The Web Services Invocation Framework (WSIF) is a WSDL-oriented Java API. You use this API to invoke Web services dynamically, regardless of the service implementation format (for example enterprise bean (EJB)) or the service access mechanism (for example Java Messaging Service (JMS)).

Using WSIF, you can move away from the usual Web services programming model of working directly with the SOAP APIs, towards a model where you interact with representations of the services. You can therefore work with the same programming model regardless of how the service is implemented and accessed.

If you want to know more about the issues that WSIF addresses, see Goals of WSIF.

If you want to know how WSIF addresses these issues, see An overview of WSIF.

To use WSIF, see the following topics:
- Using WSIF to invoke Web services.
- WSIF system management and administration.
- WSIF API.

For more information about working with WSIF, visit the Web sites listed in Web services: Resources for Learning.

Goals of WSIF

SOAP bindings for Web services are part of the WSDL specification, therefore when most developers think of using a Web service, they immediately think of assembling a SOAP message and sending it across the network to the service endpoint, using a SOAP client API. For example: using Apache SOAP the client creates and populates a Call object that encapsulates the service endpoint, the identification of the SOAP operation to invoke, the parameters to send, and so on.

While this process works for SOAP, it is limited in its use as a general model for invoking Web services for the following reasons:
- Web services are more than just SOAP services.
- Tying client code to a particular protocol implementation is restricting.
- Incorporating new bindings into client code is hard.
- Multiple bindings can be used in flexible ways.
- A freer Web services environment enables intermediaries.

The goals of the Web Services Invocation Framework (WSIF) are therefore:
- To give a binding-independent mechanism for Web service invocation.
- To free client code from the complexities of any particular protocol used to access a Web service.
- To enable dynamic selection between multiple bindings to a Web service.
- To help the development of Web service intermediaries.

© Copyright IBM Corp. 2003

WSIF - Web services are more than just SOAP services

You can deploy as a Web service any application that has a WSDL-based description of its functional aspects and access protocols. If you are using the Java 2 platform, Enterprise Edition (J2EE) environment, then the application is available over multiple transports and protocols.

For example, you can take a database-stored procedure, expose it as a stateless session bean, then deploy it into a SOAP router as a SOAP service. At each stage, the fundamental service is the same. All that changes is the access mechanism: from Java Database Connectivity (JDBC) to Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) and then to SOAP.

The WSDL specification defines a SOAP binding for Web services, but you can add binding extensions to the WSDL so that, for example, you can offer an enterprise bean as a Web service using RMI-IIOP as the access protocol. You can even treat a single Java class as a Web service, with in-thread Java method invocations as the access protocol. With this broader definition of a Web service, you need a binding-independent mechanism for service invocation.

WSIF - Tying client code to a particular protocol implementation is restricting

If your client code is tightly bound to a client library for a particular protocol implementation, it can become hard to maintain.

For example, if you move from Apache SOAP to Java Messaging Service (JMS) or enterprise bean, the process can take a lot of time and effort. To avoid these problems, you need a protocol implementation-independent mechanism for service invocation.

WSIF - Incorporating new bindings into client code is hard

As is explained in Web services are not just SOAP services, if you want to make an application that uses a custom protocol work as a Web service, you can add extensibility elements to WSDL to define the new bindings. But in practice, achieving this capability is hard. For example you have to design the client APIs to use this protocol. If your application uses just the abstract interface of the Web service, you have to write tools to generate the stubs that enable an abstraction layer. These tasks can take a lot of time and effort. What you need is a service invocation mechanism that allows you to update existing bindings, and to add new bindings.

WSIF - Multiple bindings can be used in flexible ways

Imagine that you have successfully deployed an application that uses a Web service which offers multiple bindings. For example, imagine that you have a SOAP binding for the service and a local Java binding that lets you treat the local service implementation (a Java class) as a Web service.

The local Java binding for the service can only be used if the client is deployed in the same environment as the service. In this case, it is more efficient to communicate with the service by making direct Java calls than by using the SOAP binding.

If your clients could switch the actual binding used based on run-time information, they could choose the most efficient available binding for each situation. To take advantage of Web services that offer multiple bindings, you need a service

invocation mechanism that can switch between the available service bindings at run-time, without having to generate or recompile a stub.

WSIF - Enabling a freer Web services environment promotes intermediaries

Web services offer application integrators a loosely-coupled paradigm. In such environments, intermediaries can be very powerful.

Intermediaries are applications that intercept the messages that flow between a service requester and a target Web service, and perform some mediating task (for example logging, high-availability or transformation) before passing on the message. They can be as small as a simple Web service, or as large as the Web services gateway. The Web Services Invocation Framework (WSIF) is designed to make building intermediaries both possible and simple. Using WSIF, intermediaries can add value to the service invocation without needing transport-specific programming.

An overview of WSIF

The Web Services Invocation Framework (WSIF) provides a Java API for invoking Web services, independent of the format of the service or the transport protocol through which it is invoked. This framework addresses all of the issues identified in the goals of WSIF.

WSIF provides the following features:
- An API that provides binding-independent access to any Web service.
- A close relationship with WSDL, so it can invoke any service that you can describe in WSDL.
- A stubless and completely dynamic invocation of a Web service.
- The capability to plug a new or updated implementation of a binding into WSIF at run-time.
- The option to defer the choice of a binding until run-time.

WSIF is designed to work both in an unmanaged environment (stand-alone) and inside a managed container. You can use the Java Naming and Directory Interface (JNDI) to find the WSIF service, or you can use the location described in the WSDL.

For more conceptual information about WSIF and WSDL, see the following topics:
- WSIF and WSDL
- WSIF architecture
- Using WSIF with Web services that offer multiple bindings
- WSIF usage scenarios
- Dynamic invocation

WSIF architecture

The Web Services Invocation Framework (WSIF) architecture is shown in the

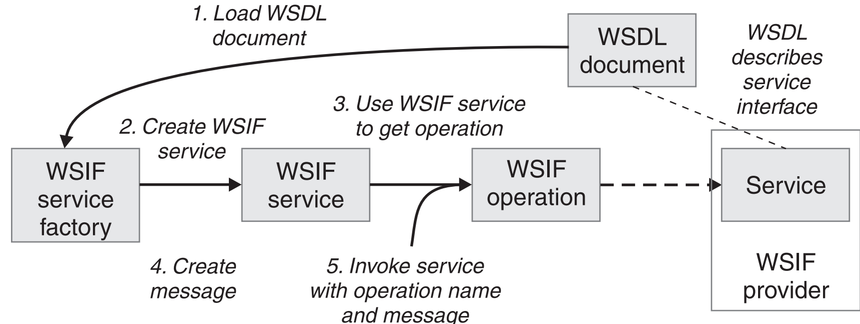


figure.

The components of this architecture include:

WSDL document
: The Web service WSDL document contains the location of the Web service. The binding document defines the protocol and format for operations and messages defined by a particular portType.

WSIF service
: The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation. For more information, see Finding a port factory or service

WSIF operation
: The run-time representation of an operation, called WSIFOperation is responsible for invoking a service based on a particular binding. For more information, see WSIF API reference: Using ports.

WSIF provider
: A WSIF provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol. WSIF includes SOAP providers, JMS providers, Java providers and EJB providers. For more information, see Using the WSIF providers.

Using WSIF with Web services that offer multiple bindings

Using WSIF, a client application can choose dynamically the optimal binding to use for invoking Web service operations.

For example, a Web service might offer a SOAP binding, and also a local Java binding so that you can treat the local service implementation (a Java class) as a Web service. If a client application is deployed in the same environment as the service, then this client can use the local Java binding for the service. This provides more efficient communication between the client and the service by making direct Java calls rather than indirect calls using the SOAP binding.

For more information on how to configure a client to dynamically select between multiple bindings, see Developing a WSIF service.

WSIF and WSDL

WSDL is the acronym for Web Services Description Language.

In WSDL a service is defined in three distinct sections:

- The **portType**. This section defines the abstract interface offered by the service. A portType defines a set of *operations*. Each operation can be In-Out (request-response), In-Only, Out-Only and Out-In (Solicit-Response). Each

operation defines the input and/or output *messages*. A message is defined as a set of *parts*, and each part has a schema-defined type.
- The **binding**. This section defines how to map between the abstract portType and a real service format and protocol. For example the SOAP binding defines the encoding style, the SOAPAction header, the namespace of the body (the targetURI), and so on.
- The **port**. This section defines the actual location (endpoint) of the available service. For example, the HTTP Web address at which a SOAP service is available.

Currently in WSDL, each port has one and only one binding, and each binding has a single portType. But (more importantly) each service (portType) can have multiple ports, each of which represents an alternative location and binding for accessing that service.

The Web Services Invocation Framework (WSIF) follows the semantics of WSDL as much as possible:
- The WSIF dynamic invocation API directly exposes run-time equivalents of the model from WSDL. For example, invocation of an operation involves executing an operation with an input message.
- WSDL has extension points that support the addition of new ports and bindings. This enables WSDL to describe new systems. The equivalent concept in WSIF is a provider, that enables WSIF to understand a class of extensions and thereby to support a new service implementation type.

As a metadata-based invocation framework, WSIF follows the design of the metadata. As WSDL is extended, WSIF is updated to follow.

The implicit and primary type system of WSIF is XML schema. WSIF supports invocation using dynamic proxies, which in turn support Java type systems, but when you use the WSIFMessage interface it is your responsibility to populate WSIFMessage objects with data based on the XML schema types as defined in the WSDL document. You should define your object types by a canonical and fixed mapping from schema types into the run-time.

For more information on WSDL, see Web services: Resources for learning.

WSIF usage scenarios

This topic describes two brief scenarios that illustrate the role WSIF plays in the emerging Web services environment.

Scenario: Redevelopment and redeployment

When you first implement a Web service, you create a simple prototype. When you want to move a prototype Web service into production, you often need to redevelop and redeploy it.

The Web Services Invocation Framework (WSIF) uses the same API calls irrespective of the underlying technologies, therefore if you use WSIF:
- You can reimplement and redeploy your services without changing the client code.
- You can use existing reliable and high-performance infrastructures like Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) and Java Messaging Service (JMS) without sacrificing the location-independence that the Web service model offers.

Scenario: Service Flow composition

A service flow typically invokes a Web service, then passes the response from one Web service to the next Web service, perhaps performing some transformation in the middle.

There are two key aspects to this flow that WSIF provides:
- A representation of the service invocation based on the metadata in WSDL.
- The ability to build invocations based solely on the portType, which can therefore be used in any implementation.

For example, imagine that you build a meta-service that uses a number of services to build a process. Initially, several of those services are simple Java bean prototypes that are written and exposed through SOAP, but you plan to reimplement some of them as EJB components, and to out-source others.

If you use SOAP, it ties up multiple threads for every onward invocation, because they pass through the Web server and servlet engine and on to the SOAP router. If you use WSIF to call the beans directly, you get much better performance compared to SOAP and you do not lose access or location transparency. Using WSIF, you can replace the Java bean implementations with EJB implementations without changing the client code. To move some of the Web services from local implementations to external SOAP services, you just update the WSDL.

Dynamic invocation

For the Web Services Invocation Framework (WSIF), dynamic invocation means providing the following levels of support when invoking Web services:
1. Support for WSDL extensions and bindings that were not known at build time.
2. Support for Web services that were not known at build time.

WSIF supports (1) through the use of providers.

The providers support (2) by using the WSDL description to access the target service.

Using WSIF to invoke Web services

You invoke a Web service dynamically by using the WSIF API directly.

You only specify the location of the WSDL file for the service, the name of the operation to invoke, and any operation arguments. All the information needed to access the Web service (the abstract interface, the binding, and the service endpoint) is available through the WSDL.

This kind of invocation does not require stub classes and does not need a separate compilation cycle.

More information on using the Web Services Invocation Framework (WSIF) to invoke Web services is provided in the following topics:
- Using the WSIF providers.
- Developing a WSIF service.
- Using complex types.
- Using the Java Naming and Directory Interface (JNDI).
- 5.0.2 + Passing SOAP messages with attachments using WSIF.
- Interacting with the J2EE container in WebSphere Application Server.

- Running WSIF as a client.

Using the WSIF providers

A Web Services Invocation Framework (WSIF) provider is an implementation of a WSDL binding that can run a WSDL operation through a binding-specific protocol.

Providers implement the interface between the WSIF API and the actual implementation of a service. Providers are pluggable within the WSIF framework, and are registered according to the namespace of the WSDL extension that they implement. Some providers use the Java 2 platform, Enterprise Edition (J2EE) programming model to utilize J2EE services. If a provider is available, but its required class libraries are not, then the provider is disabled.

WebSphere Application Server includes the following WSIF providers:
- SOAP (over HTTP) provider.
- JMS providers (SOAP over JMS, and native JMS).
- Java provider.
- EJB provider.

Note:
-
-

Using the SOAP provider

The SOAP provider allows WSIF stubs and dynamic clients to invoke SOAP services.

The Web Services Invocation Framework (WSIF) SOAP provider supports SOAP 1.1 over HTTP. The SOAP provider uses Apache SOAP 2.3 for parsing and creating SOAP messages, but it is not limited to invoking services from Apache SOAP.

The WSIF SOAP provider supports:
- SOAP-ENC encoding.
- 5.0.1 Remote Procedure Call (RPC) style SOAP messages.
- 5.0.2 + RPC style and Document style SOAP messages.
- 5.0.2 + SOAP messages with attachments.

The SOAP provider is not transactional.

If you have a Web service that you expect multiple clients to use connecting over SOAP, then before you deploy the service you must set up your application deployment descriptor file `dds.xml` to handle multiple connections correctly. For more information, see WSIF troubleshooting tips.

For an example of the sort of code changes that need to be made in the WSDL file for a SOAP provider, see the following topics:
- The SOAP over JMS provider - writing the WSDL extension.
- 5.0.2 + SOAP messages with attachments - Writing the WSDL extensions.

Using the JMS providers

The JMS providers enable a WSIF service to be invoked through JMS.

The Java Messaging Service (JMS) is an API for transport technology. The mapping to a JMS destination is defined during deployment and maintained by the container.

The JMS destination endpoint for a Web service can be realized in any of the following ways:
- The JMS destination for the queue can be the Web service implementation.
- The JMS destination can be (but is not required to be) associated with a message-driven bean by the EJB container, thereby allowing the message-driven bean to be the Web service implementation.
- (For SOAP over JMS) The JMS destination can unwrap the JMS message and route the SOAP message to a Web service that is implemented as a stateless session bean.

The JMS destination endpoint must respect the interaction model expected by the client and defined by the WSDL. It must return a response if one is required.

When the JMS destination endpoint creates the JMS response message the following rules must be followed:
- The response message must be sent to `JMSReplyTo` from the incoming request.
- The `JMSCorrelationID` value of the response message must be set to the `JMSMessageID` value from the request message.
- The response must be sent with a `deliveryMode` value equal to the `JMSDeliveryMode` value of the request message.
- The response must be sent with a `priority` value equal to the `JMSPriority` value of the request message.
- The `timetolive/JMSExpiration` value must be set to a value that equals the `JMSExpiration` value of the request message.

The client does not see any of these headers. The container receives the JMS message and (for SOAP over JMS) removes the SOAP message to send to the client.

See also the following topics:
- Using the SOAP over JMS provider
- Using the native JMS provider
- The JMS providers - Configuring the client and server

Using the SOAP over JMS provider:

For information on working with the Java Messaging Service (JMS) API, see Using the JMS providers.

The SOAP message, including the SOAP envelope, is wrapped with a JMS message and put on the appropriate queue. The container receives the JMS message and removes the SOAP message to send to the client.

For detailed implementation information, see the following topics:
- The SOAP over JMS provider - writing the WSDL extension
- The JMS providers - Configuring the client and server

The SOAP over JMS provider - Writing the WSDL extension:

If a SOAP message contains only XML, then it can be carried on the Java Messaging Service (JMS) transport with the JMS message body type **TextMessage**.

The WSDL binding extension for SOAP over JMS varies only slightly from the SOAP over HTTP binding.

Selecting the SOAP over JMS binding

You set the `transport` attribute of the `<soap:binding>` tag to indicate that JMS is used. If you also set the `style` attribute to `rpc` (Remote Procedure Call), then the Web Services Invocation Framework (WSIF) assumes that an operation is invoked on the Web service endpoint:

```
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/jms"/>
```

Setting the JMS address

For SOAP over JMS, the `<wsdl:port>` tag must contain a `<jms:address>` element. This element provides the information required for a client to connect correctly to the Web service using the JMS programming model. Typically, it is the stubs generated to support the SOAP over JMS binding that act as the JMS client. Alternatively, the Web service client can use the JMS programming model directly.

The `<jms:address>` element takes this form:

```
<jms:address

    destinationStyle="queue"
    jmsVendorURI="http://ibm.com/ns/mqseries"?
    initialContextFactory="com.ibm.NamingFactory"?
    jndiProviderURL="iiop://something:900/wherever"?
    jndiConnectionFactoryName="orange"
    jndiDestinationName="fred"
/>
```

where attributes marked with a question mark (?) are optional.

The optional `jmsVendorURI` attribute is a string that uniquely identifies the JMS implementation. WSIF ignores this URI, which is used by the client developer and perhaps the client implementation to determine if it has access to the correct JMS provider in the client run-time.

The optional attributes `initialContextFactory` and `jndiProviderURL` can only be omitted if the run-time has a default Java Naming and Directory Interface (JNDI) provider configured.

The `jndiConnectionFactoryName` attribute gives the name of a JMS ConnectionFactory object, which can be looked up within the JNDI context given by the jndiContext attribute. This ConnectionFactory object is used to create a JMS connection to the JMS provider instance that owns the queue. In a simple configuration, the same ConnectionFactory object is used by the server message listener and by the clients. However the server and the clients can use different ConnectionFactory objects, provided that they all create connections to the same JMS provider instance.

Setting the JMS headers and properties

You use the `<jms:property>` tag to set the JMS headers and properties. This tag maps either a message part, or a literal value, into a JMS property:

```
<jms:property name="Priority" {part="requestPriority" | value="fixedValue"}/>
```

If the `<jms:property>` has a literal value, then it can also be nested within the `<jms:address>` tag:

```
<jms:property name="Priority" value="fixedValue" />
```

This form of the <jms:property> tag is also used in the native JMS binding.

Here is an example of a WSDL that defines a SOAP over JMS binding:

```
<!-- Example: SOAP over JMS Text Message -->

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
        name="StockQuoteInterfaceDefinitions"
        targetNamespace="urn:StockQuoteInterface"
        xmlns:tns="urn:StockQuoteInterface"
        xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <wsdl:message name="GetQuoteInput">
        <part name="symbol" type="xsd:string"/>
    </wsdl:message>
    <wsdl:message name="GetQuoteOutput">
        <part name="value" type="xsd:float"/>
    </wsdl:message>

    <wsdl:portType name="StockQuoteInterface">
        <wsdl:operation name="GetQuote">
            <wsdl:input message="tns:GetQuoteInput"/>
            <wsdl:output message="tns:GetQuoteOutput"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="StockQuoteSoapJMSBinding" type="tns:StockQuoteInterface">
        <soap:binding style="rpc"
                 transport="http://schemas.xmlsoap.org/soap/jms"/>
        <wsdl:operation name="GetQuote">
            <soap:operation soapAction="urn:StockQuoteInterface#GetQuote"/>
            <wsdl:input>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="encoded" namespace="urn:StockQuoteService"
                     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="StockQuoteService">
        <wsdl:port name="StockQuoteServicePort"
                      binding="sqi:StockQuoteSoapJMSBinding">
            <jms:address destinationStyle="queue"
                     jndiConnectionFactoryName="myQCF"
                     jndiDestinationName="myQ"
                     initialContextFactory="com.ibm.NamingFactory"
                     jndiProviderURL="iiop://something:900/" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Using the native JMS provider:

Using the native JMS provider, WSIF clients can treat a JMS destination as a Web service.

For information on working with the Java Messaging Service (JMS) API, see Using the JMS providers.

For detailed implementation information, see the following topics:
- The native JMS provider - Writing the WSDL extension

- The JMS providers - Configuring the client and server

The native JMS provider - Writing the WSDL extension:

The WSDL extensions for the Java Messaging Service (JMS) are identified with the namespace prefix `jms`. For example, `<jms:binding>`.

Operations

The supported operations are either one-way operations (`send` for JMS point-to-point messaging, or `publish` for JMS publish and subscribe messaging) or request-response operations (`send` and `receive` for JMS point-to-point messaging). The WSDL operations therefore specify either an input message only, or an input and an output message.

Fault messages

Operations that describe message interfaces with a native JMS binding do not have fault messages. No assumptions are made about the message schema or the semantics of message properties, therefore no distinction can be made between output and fault messages.

Setting the JMS message body type

You use the `<jms:binding>` extension to specify the JMS message body type:

```
<wsdl:binding ... >
   <jms:binding type="messageBodyType" />
    ...
</wsdl:binding>
```

where *messageBodyType* is either `ObjectMessage` or `TextMessage`.

Specifying the parts to use for the input and output messages

For JMS text messages and JMS object messages created from one or more WSDL message parts, you use the `<jms:input>` and `<jms:output>` extensions to specify the message parts to use for the JMS messages:

```
<wsdl:input ... >
   <jms:input parts="part1 part2 ..." />
</wsdl:input>

<wsdl:output ... >
   <jms:output parts="part1 part2 ..." />
</wsdl:output>
```

In the next example, the WSDL message has just one part that contains the complete message body. This message body might result from a mapping of some other representation (see **Mapping data types**).

```
<wsdl:input ... >
   <jms:input parts="part1" />
</wsdl:input>
```

If no parts are defined, then all the message parts are used.

Mapping data types

You use the `<format>` extensions to map data types:

```
<wsdl:binding ... >
   <jms:binding type="..." />

   <format:typeMapping encoding="Java" style="Java">
```

```
      <format:typeMap typeName="..." formatType="targetType"/>
   </format:typemapping>
   ...
</wsdl:binding>
```

The value of *targetType* is dependent on the JMS message body type (see **Setting the JMS message body type**). For JMS object messages, the target data type implements the java.io.Serializable class. For JMS text messages, the target data type is always java.lang.String.

The <format> extensions are also used in other bindings that deal with Java interfaces.

Setting the JMS headers and properties

JMS does not make assumptions about message headers. For example, if the JMS provider is MQSeries then each JMS message carries an RFH2 header. However you can access data in this message header indirectly, by getting and setting JMS message properties.

When you want your application to pass a property into the Web Services Invocation Framework (WSIF) as a part on the WSIF message, you use a <jms:property> tag. When you want to hard code an actual property value into the WSDL, you use a <jms:propertyValue> tag. The <jms:propertyValue> tag contains a specification of a literal value and its associated XML schema type.

You can specify <jms:property> and <jms:propertyValue> extensions within the <wsdl:input> tag in the binding operation, and also within the <jms:address> tag. For the<wsdl:output> tag in the binding operation, you can only specify the <jms:property> extension. Property values that are set in the <jms:property> tag take precedence over values set in the <<jms:propertyValue> tag, and property values that are set in the binding operation (in the <input> and <output> tags) take precedence over values set in the <jms:address> tag.

Here is an example of the <jms:property> and <jms:propertyValue> tags nested within the <input> and <output> tags:

```
<wsdl:input ... >

    <jms:property name="propertyName" part="partName" />

    <jms:propertyValue name="propertyName"
               type="xsdType" value="actualValue" />

</wsdl:input>

<wsdl:output ... >

    <jms:property name="propertyName" part="partName" />

</wsdl:output>
```

where *propertyName* identifies the JMS property that is associated with the header field, and *partName* identifies the message part that is associated with the property.

The JMS property identified by *propertyName* can be user-defined, or it can be one of the following predefined JMS message header fields:

Value	Java type
JMSMessageId	java.lang.String
JMSTimeStamp	long
JMSCorrelationId	byte [ ] or java.lang.String
JMSReplyTo	javax.jms.Destination
JMSDestination	javax.jms.Destination
JMSDeliveryMode	int
JMSRedelivered	boolean
JMSType	java.lang.String
JMSExpiration	long

See the JMS specification for restrictions that apply when setting JMS header field values. Attempts to set restricted values are ignored.

For application-defined JMS message properties, the Java types used in the native JMS binding implementation (used for calls to the corresponding JMS methods) are derived from the XML schema type in the abstract interface (`<wsdl:part>` tag), and from the type mapping information in the format binding (`<format:typemap>` tag).

Handling transactions

Independent of other JMS properties, the asynchronous processing of request-response operations has implications for callers running in a transaction scope. The send request part and the receive response part are separated into two transactions, because the send needs to be committed in order for the request message to become visible. Implementations that process WSDL for asynchronous request-response operations (such as WSIF) must therefore take the following additional actions:

- They must ensure that the send request transaction returns a correlation ID to the user, and provides a **callback** with which users can pass in the response message to process the receive response transaction.
- They might implement their own response message "listener" in order to recognize the arrival of response messages, and to manage the correlation to the request message.

The JMS text message contains a **java.lang.String**. In this example, the WSDL message contains only one part that represents the whole message body:

```
<!-- Example 1: JMS Text Message -->

<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest"> ... </wsdl:message>
   <wsdl:message name="JmsOperationResponse"> ... </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
```

```
   </wsdl:portType>

   <wsdl:binding name="JmsBinding" type="JmsPortType">
      <jms:binding type="TextMessage" />

      <format:typemapping style="Java" encoding="Java">
         <format:typemap name="xsd:String" formatType="String" />
      </format:typemapping>

      <wsdl:operation name="JmsOperation">
         <wsdl:input message="JmsOperationRequest">
            <jms:input parts="requestMessageBody" />
         </wsdl:input>
         <wsdl:output message="JmsOperationResponse">
            <jms:output parts="responseMessageBody" />
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="JmsService">
      <wsdl:port name="JmsPort" binding="JmsBinding">
         <jms:address destinationStyle="queue"
                      jndiConnectionFactoryName="myQCF"
                      jndiDestinationName="myDestination"/>
      </wsdl:port>
   </wsdl:service>

</wsdl:definitions>
```

As an extension to the previous JMS message example, the following example WSDL describes a request-response operation in which specific JMS property values of the request and response message are set for the request message and retrieved from the response message.

The JMS properties in the request message are set according to the values in the input message. Likewise, selected JMS properties of the response message are copied to the corresponding values of the output message. The direction of the mapping is determined by the appearance of the `<jms:property>` tag in the input or output section, respectively.

```
<!-- Example 2: JMS Message with JMS Properties -->

<wsdl:definitions ... >

   <!-- simple or complex types for input and output message -->
   <wsdl:types> ... </wsdl:types>

   <wsdl:message name="JmsOperationRequest">
      <wsdl:part name="myInt" type="xsd:int"/>
      ...
   </wsdl:message>

   <wsdl:message name="JmsOperationResponse">
      <wsdl:part name="myString" type="xsd:String"/>
      ...
   </wsdl:message>

   <wsdl:portType name="JmsPortType">
      <wsdl:operation name="JmsOperation">
         <wsdl:input name="Request"
                     message="tns:JmsOperationRequest"/>
         <wsdl:output name="Response"
                      message="tns:JmsOperationResponse"/>
      </wsdl:operation>
   </wsdl:portType>
```

```
  <wsdl:binding name="JmsBinding" type="JmsPortType">
     <!-- the JMS message type may be any of the above -->
     <jms:binding type="..." />

     <format:typemapping style="Java" encoding="Java">
        <format:typemap name="xsd:int" formatType="int" />
        ...
     </format:typemapping>

     <wsdl:operation name="JmsOperation">
        <wsdl:input message="JmsOperationRequest">
           <jms:property message="tns:JmsOperationRequest" parts="myInt" />
           <jms:propertyValue name="myLiteralString"
                         type="xsd:string" value="Hello World" />
           ...
        </wsdl:input>
        <wsdl:output message="JmsOperationResponse">
           <jms:property message="tns:JmsOperationResponse" parts="myString" />
           ...
        </wsdl:output>
     </wsdl:operation>
  </wsdl:binding>

  <wsdl:service name="JmsService">
     <wsdl:port name="JmsPort" binding="JmsBinding">
        <jms:address destinationStyle="queue"
                     jndiConnectionFactoryName="myQCF"
                     jndiDestinationName="myDestination"/>
   </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

The JMS providers - Configuring the client and server:

This topic assumes that you installed a Java Messaging Service (JMS) provider when you installed WebSphere Application Server (either the JMS provider that is embedded in WebSphere Application Server, or another provider such as WebSphere MQ). If not, install one now as described in Installing and configuring a JMS provider.

To enable a service to be invoked through JMS by a Web Services Invocation Framework (WSIF) client application, complete the following steps:

1. Use the administrative console to create and configure a queue connection factory and a queue destination as described in Configuring JMS provider resources.
2. Use the administrative console to add the new queue destination to the list of JMS Server destination names for your application server as described in Managing WebSphere internal JMS servers. Ensure that the Initial State is started.
3. Put the JNDI names of the queue destination and queue connection factory, as well as your JNDI configuration, in the WSDL file.

You should also understand the specific ways in which WSIF interacts with JMS:

- Only input JMS properties are supported.
- WSIF needs two queues when invoking an operation: one for the request message and one for the reply. The replyTo queue is by default a temporary queue, which WSIF creates on behalf of the application. You can specify a permanent queue by setting the JMSReplyTo property to the JNDI name of a queue.

- WSIF uses the default values for properties set by the JMS implementation. However in MQSeries and in some other JMS implementations, messages are persistent by default, and the default temporary queue is of type temporary dynamic and cannot have persistent messages written to it. Therefore your JMS listener can fail to write a persistent response message to the temporary replyTo queue.

  Note: If you are using MQSeries, you need to create a temporary model queue that is of type permanent dynamic, then pass this model as the tempmodel of your queue connection factory. This will ensure that persistent messages are written to a temporary replyTo queue that is of type permanent dynamic.

Using the Java provider

Using the WSIF Java provider, WSIF can invoke Java code.

This means that, in a thin-client environment such as a Java Virtual Machine (JVM) or Tomcat test run-time, you can define shortcuts to local Java programs.

The Web Services Invocation Framework (WSIF) Java provider is not intended for use in a Java 2 platform, Enterprise Edition (J2EE) environment. There is a difference between a client using the WSIF Java provider to invoke a Java component, and implementing a Web service as a Java component on the server side.

The Java binding exploits the format binding for type mapping. Using the format binding, your WSDL can define the mapping between XML schema types and Java types.

The Java provider requires that the targeted Java classes reside in the class path of the client. The Java method is invoked synchronously, in-process, in-thread, with the current thread and Object Request Broker (ORB) contexts.

The Java provider is not transactional.

For examples of the code changes that need to be made in the WSDL file, see The Java provider - Writing the WSDL extension.

The Java provider - Writing the WSDL extension:

The Java provider supports the invocation of a method on a local Java object.

To use the Java provider, you need the following binding specified in the WSDL:

```
<!-- Java binding -->
<binding .... >
    <java:binding />
    <format:typeMapping style="Java" encoding="Java"/>?
        <format:typeMap name="qname" formatType="nmtoken"/>*
    </format:typeMapping>
    <operation>*
        <java:operation
            methodName="nmtoken"
            parameterOrder="nmtoken"
            returnPart="nmtoken"?
            methodType="instance|constructor" />
        <input name="nmtoken"? />?
```

```
            <output name="nmtoken"? />?
            <fault name="nmtoken"? />?
        </operation>
    </binding>
```

In this example:
- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The name attribute of the <format:typeMap> element is a qualified name of a simple or complex type used by one of the Java operations.
- The formatType attribute of the <format:typeMap> element is the fully qualified class name for the Java class to which the element specified by name maps.
- The methodName attribute of the <java:operation> element is the name of the method on the Java object that is called by the operation.
- The parameterOrder attribute of the <java:operation> element contains a white space-separated list of part names that define the order in which they are passed to the Java object method.
- The methodType attribute of the <java:operation> element must be set to either instance or constructor. The value specifies whether the method that is invoked on the object is an instance method or a constructor for the object.

In the next example, the className attribute of the <java:address> element specifies the fully qualified class name of the object containing the method to invoke:

```
    <service ... >
        <port>*
            <java:address
                className="nmtoken"/>
        </port>
  </service>
```

Using the EJB provider

Using the EJB provider, WSIF clients can invoke enterprise beans.

The EJB client JAR file must be available in the client run-time with the current provider. The enterprise bean is invoked using normal EJB invocation methods, using Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP), with the current security and transaction contexts. If the EJB provider is invoked within a transaction, the transaction is passed to the onward service and the standard EJB transaction attribute applies.

If there are multiple implementations of the service, it is up to the service providers to make sure that every implementation offers the same semantics. For example, in the case of transactions, the bean deployer must specify TX_REQUIRES_NEW to force a new transaction.

For examples of the code changes that need to be made in the WSDL file, see The EJB provider - Writing the WSDL.

The EJB provider - Writing the WSDL extension:

The EJB provider supports the invocation of an enterprise bean through Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP).

To use the EJB provider, you need the following binding specified in the WSDL:

```
    <!-- EJB binding -->
    <binding .... >
        <ejb:binding />
        <format:typeMapping style="Java" encoding="Java"/>?
```

```
            <format:typeMap name="qname" formatType="nmtoken"/>*
        </format:typeMapping>
        <operation>*
            <ejb:operation
                methodName="nmtoken"
                parameterOrder="nmtoken"
                returnPart="nmtoken"?
                interface="remote|home" />
            <input name="nmtoken"? />?
            <output name="nmtoken"? />?
            <fault name="nmtoken"? />?
        </operation>
    </binding>
```

In this example:
- A question mark (?) means optional, and an asterisk (*) means 0 or more.
- The name attribute of the <format:typeMap> element is a qualified name of a simple or complex type used by one of the EJB operations.
- The formatType attribute of the <format:typeMap> element is the fully qualified class name for the Java class to which the element specified by name maps.
- The methodName attribute of the <ejb:operation> element is the name of the method on the enterprise bean that is called by the operation.
- The parameterOrder attribute of the <ejb:operation> element contains a white space-separated list of part names that define the order in which they are passed to the EJB method.
- The interface attribute of the <ejb:operation> element must be set to either remote or home. The value specifies the interface of the enterprise bean on which the method named by the methodName attribute is accessible.

In the next example:
- The className attribute of the <ejb:address> element specifies the fully qualified class name of the home interface class of the enterprise bean.
- The jndiName attribute of the <ejb:address> element specifies the full Java Naming and Directory Interface (JNDI) name that is used to look up the enterprise bean.
- The initialContextFactory attribute of the <ejb:address> element is optional and specifies the initial context factory class.
- The jndiProviderURL attribute of the <ejb:address> element is optional and specifies the JNDI provider Web address.

```
<service ... >
    <port>*
        <ejb:address
            className="nmtoken"
            jndiName="nmtoken"
            initialContextFactory="nmtoken" ?
            jndiProviderURL="nmtoken" ? />
    </port>
</service>
```

Developing a WSIF service

A Web Services Invocation Framework (WSIF) service is a Web service that uses WSIF.

To develop a WSIF service, develop the Web service (or use an existing Web service), then develop the WSIF client based on the WSDL document for that Web service.

There are also two pre-built WSIF Samples available for download from the Samples Central page of the IBM WebSphere Developer Domain Web site:
- The Address Book Sample.
- The Stock Quote Sample.

For more information on using the pre-built Samples, see the documentation that is included in the download package.

To develop a WSIF service, complete the following steps:
1. Develop the Web service.

   Use Web services tools to discover, create, and publish the Web service. You can develop Java bean, enterprise bean, and URL Web services. You can use Web service tools to create skeleton Java code and a sample application from a WSDL document. For example, an enterprise bean can be offered as a Web service, using Remote Method Invocation over Internet Inter-Orb Protocol (RMI-IIOP) as the access protocol. Or you can use a Java class as a Web service, with native Java invocations as the access protocol.

   You can use the WebSphere Studio Application Developer to create a Web service from a Java application, as described in its StockQuote service tutorial. The Java application that you use in this scenario returns the last trading price from the Internet Web site www.xmltoday.com, given a stock symbol. Using the Web service wizard, you generate a binding WSDL document named `StockQuoteService-binding.wsdl` and a service WSDL document named `StockQuoteService-service.wsdl` from the `StockQuoteService.java` bean. You then deploy the Web service to a Web server, generate a client proxy to the Web service, and generate a sample application that accesses the StockQuoteService through the client proxy. You test the StockQuote Web service, publish it using the IBM UDDI Explorer, and then discover the StockQuote Web service in the IBM UDDI Test Registry.
2. Develop the WSIF client. The information you need to develop a WSIF client is provided in the following topics:
   - Developing the WSIF client - the Address Book Sample gives example code to show how you define a Web service in WSDL.
   - Using the WSIF providers describes the available providers, and gives example code of how their WSDL extensions are coded.
   - WSIF API defines the main interfaces that your client uses to support the invocation of Web services defined in WSDL.

   The Address Book Sample is written for synchronous interaction. If you are using a JMS provider, your WSIF client might need to act asynchronously. WSIF provides two main features that meet this requirement:
   - A **correlation service** that assigns identifiers to messages so that the request can match up with the (eventual) response.
   - A **response handler** that picks up the response from the Web service at a later time.

   For more information, see the WSIF API topic WSIFOperation - Asynchronous interactions reference.

Developing the WSIF client - the Address Book Sample

The code fragments in this topic show you how to use the Web Services Invocation Framework (WSIF) API to invoke the AddressBook Sample Web service dynamically.

This is example code for dynamic invocation of the AddressBook sample Web service using WSIF:

```
  try {
      String wsdlLocation="clients/addressbook/AddressBookSample.wsdl";

      // The starting point for any dynamic invocation using wsif is a
      // WSIFServiceFactory. We create ourselves one via the newInstance
      // method.
      WSIFServiceFactory factory = WSIFServiceFactory.newInstance();

      // Once we have a factory, we can use it to create a WSIFService object
      // corresponding to the AddressBookService service in the wsdl file.
      // Note: since we only have one service defined in the wsdl file, we
      // do not need to use the namespace and name of the service and can pass
      // null instead. This also applies to the port type, although values have
      // been used below for illustrative purposes.
      WSIFService service = factory.getService(
          wsdlLocation,    // location of the wsdl file
          null,            // service namespace
          null,            // service name
         "http://www.ibm.com/namespace/wsif/samples/ab", // port type namespace
          "AddressBookPT" // port type name
      );

      // The AddressBook.wsdl file contains the definitions for two complexType
      // elements within the schema element. We will now map these complexTypes
      // to Java classes. These mappings are used by the Apache SOAP provider
      service.mapType(
          new javax.xml.namespace.QName(
              "http://www.ibm.com/namespace/wsif/samples/ab/types",
              "address"),
          Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"));
      service.mapType(
          new javax.xml.namespace.QName(
              "http://www.ibm.com/namespace/wsif/samples/ab/types",
              "phone"),
          Class.forName("com.ibm.www.namespace.wsif.samples.ab.types.WSIFPhone"));
          // We now have a WSIFService object. The next step is to create a WSIFPort
          // object for the port we wish to use. The getPort(String portName) method
          // allows us to generate a WSIFPort from the port name.
      WSIFPort port = null;

      if (portName != null) {
          port = service.getPort(portName);
      }
      if (port == null) {
          // If no port name was specified, attempt to create a WSIFPort from
          // the available ports for the port type specified on the service
          port = getPortFromAvailablePortNames(service);
      }

      // Once we have a WSIFPort, we can create an operation. We are going to execute
      // the addEntry operation and therefore we attempt to create a WSIFOperation
      // corresponding to it. The addEntry operation is overloaded in the wsdl ie.
// there are two versions of it, each taking different parameters (parts).
// This overloading requires that we specify the input and output message
// names for the operation in the createOperation method so that the correct
// operation can be resolved.
      // Since the addEntry operation has no output message, we use null for its name.
      WSIFOperation operation =
          port.createOperation("addEntry", "AddEntryWholeNameRequest", null);

      // Create messages to use in the execution of the operation. This should
      // be done by invoking the createXXXXXMessage methods on the WSIFOperation.
      WSIFMessage inputMessage = operation.createInputMessage();
      WSIFMessage outputMessage = operation.createOutputMessage();
      WSIFMessage faultMessage = operation.createFaultMessage();

      // Create a name and address to add to the addressbook
```

```
    String nameToAdd="Chris P. Bacon";
    WSIFAddress addressToAdd =
        new WSIFAddress (1,
            "The Waterfront",
            "Some City",
            "NY",
            47907,
            new WSIFPhone (765, "494", "4900"));

    // Add the name and address to the input message
    inputMessage.setObjectPart("name", nameToAdd);
    inputMessage.setObjectPart("address", addressToAdd);

    // Execute the operation, obtaining a flag to indicate its success
    boolean operationSucceeded =
        operation.executeRequestResponseOperation(
            inputMessage,
            outputMessage,
            faultMessage);

    if (operationSucceeded) {
        System.out.println("Successfully added name and address to addressbook\n");
    } else {
        System.out.println("Failed to add name and address to addressbook);
    }

    // Start from fresh
    operation = null;
    inputMessage = null;
    outputMessage = null;
    faultMessage = null;

    // This time we will lookup an address from the addressbook.
    // The getAddressFromName operation is not overloaded in the
    // wsdl and therefore we can simply specify the operation name
    // without any input or output message names.
    operation = port.createOperation("getAddressFromName");

    // Create the messages
    inputMessage = operation.createInputMessage();
    outputMessage = operation.createOutputMessage();
    faultMessage = operation.createFaultMessage();

    // Set the name to find in the addressbook
    String nameToLookup="Chris P. Bacon";
    inputMessage.setObjectPart("name", nameToLookup);

    // Execute the operation
    operationSucceeded =
        operation.executeRequestResponseOperation(
            inputMessage,
            outputMessage,
            faultMessage);

    if (operationSucceeded) {
        System.out.println("Successful lookup of name '"+nameToLookup+"' in addressbook");

        // We can obtain the address that was found by querying the output message
        WSIFAddress addressFound = (WSIFAddress) outputMessage.getObjectPart("address");
        System.out.println("The address found was:");
        System.out.println(addressFound);
    } else {
        System.out.println("Failed to lookup name in addressbook");
    }

} catch (Exception e) {
```

```
        System.out.println("An exception occurred when running the sample: ");
        e.printStackTrace();
    }
}
```

The preceding code refers to the following Sample method:

```
    WSIFPort getPortFromAvailablePortNames(WSIFService service)
            throws WSIFException {
        String portChosen = null;

        // Obtain a list of the available port names for the service
        Iterator it = service.getAvailablePortNames();
        {
            System.out.println("Available ports for the service are: ");
            while (it.hasNext()) {
                String nextPort = (String) it.next();
                if (portChosen == null)
                    portChosen = nextPort;
                System.out.println(" - " + nextPort);
            }
        }
        if (portChosen == null) {
            throw new WSIFException("No ports found for the service!");
        }
        System.out.println("Using port " + portChosen + "\n");

        // An alternative way of specifying the port to use on the service
        // is to use the setPreferredPort method. Once a preferred port has
        // been set on the service, a WSIFPort can be obtained via getPort
        // (no arguments). If a preferred port has not been set and more than
        // one port is available for the port type specified in the WSIFService,
        // an exception is thrown.
        service.setPreferredPort(portChosen);
        WSIFPort port = service.getPort();
        return port;
    }
```

The Web service uses the following classes:

WSIFAddress:

```
public class WSIFAddress implements Serializable {

    //instance variables
    private int streetNum;
    private java.lang.String streetName;
    private java.lang.String city;
    private java.lang.String state;
    private int zip;
    private WSIFPhone phoneNumber;

    //constructors
    public WSIFAddress () { }

    public WSIFAddress (int streetNum,
                        java.lang.String streetName,
                        java.lang.String city,
                        java.lang.String state,
                        int zip,
                        WSIFPhone phoneNumber) {
          this.streetNum = streetNum;
          this.streetName = streetName;
          this.city = city;
          this.state = state;
          this.zip = zip;
          this.phoneNumber = phoneNumber;
```

```
    }

    public int getStreetNum() {
          return streetNum;
    }

    public  void setStreetNum(int streetNum) {
          this.streetNum = streetNum;
    }

    public java.lang.String getStreetName() {
          return streetName;
    }

    public  void setStreetName(java.lang.String streetName) {
          this.streetName = streetName;
    }

    public java.lang.String getCity() {
          return city;
    }

    public  void setCity(java.lang.String city) {
          this.city = city;
    }

    public java.lang.String getState() {
          return state;
    }

    public  void setState(java.lang.String state) {
          this.state = state;
    }

    public int getZip() {
          return zip;
    }

    public  void setZip(int zip) {
          this.zip = zip;
    }

    public WSIFPhone getPhoneNumber() {
          return phoneNumber;
    }

    public  void setPhoneNumber(WSIFPhone phoneNumber) {
          this.phoneNumber = phoneNumber;
    }
}
```

WSIFPhone:

```
public class WSIFPhone implements Serializable {

    //instance variables
    private int areaCode;
    private java.lang.String exchange;
    private java.lang.String number;

    //constructors
    public WSIFPhone () { }

    public WSIFPhone (int areaCode,
                      java.lang.String exchange,
                      java.lang.String number) {
          this.areaCode = areaCode;
```

```
                this.exchange = exchange;
                this.number = number;
            }

            public int getAreaCode() {
                return areaCode;
            }

            public  void setAreaCode(int areaCode) {
                this.areaCode = areaCode;
        }

            public java.lang.String getExchange() {
                return exchange;
            }

            public  void setExchange(java.lang.String exchange) {
                this.exchange = exchange;
            }

            public java.lang.String getNumber() {
                return number;
            }

            public  void setNumber(java.lang.String number) {
                this.number = number;
            }
        }
```

WSIFAddressBook:

```
public class WSIFAddressBook {
    private Hashtable name2AddressTable = new Hashtable();

    public WSIFAddressBook() {
    }

    public void addEntry(String name, WSIFAddress address)
    {
        name2AddressTable.put(name, address);
    }

    public void addEntry(String firstName, String lastName, WSIFAddress address)
    {
        name2AddressTable.put(firstName+"  "+lastName, address);
    }

    public WSIFAddress getAddressFromName(String name)
        throws IllegalArgumentException
    {

        if (name == null)
        {
            throw new IllegalArgumentException("The name argument must not be " +
                                               "null.");
        }
        return (WSIFAddress)name2AddressTable.get(name);
    }

}
```

The following code is the corresponding WSDL file for the Web service:

```
<?xml version="1.0" ?>

<definitions targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:tns="http://www.ibm.com/namespace/wsif/samples/ab"
             xmlns:typens="http://www.ibm.com/namespace/wsif/samples/ab/types"
```

```
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
           xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
           xmlns:java="http://schemas.xmlsoap.org/wsdl/java/"
           xmlns:ejb="http://schemas.xmlsoap.org/wsdl/ejb/"
           xmlns="http://schemas.xmlsoap.org/wsdl/">

<types>
  <xsd:schema
     targetNamespace="http://www.ibm.com/namespace/wsif/samples/ab/types"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:complexType name="phone">
      <xsd:element name="areaCode" type="xsd:int"/>
      <xsd:element name="exchange" type="xsd:string"/>
      <xsd:element name="number" type="xsd:string"/>
    </xsd:complexType>

    <xsd:complexType name="address">
      <xsd:element name="streetNum" type="xsd:int"/>
      <xsd:element name="streetName" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:int"/>
      <xsd:element name="phoneNumber" type="typens:phone"/>
    </xsd:complexType>

  </xsd:schema>
</types>

<message name="AddEntryWholeNameRequestMessage">
  <part name="name" type="xsd:string"/>
  <part name="address" type="typens:address"/>
</message>

<message name="AddEntryFirstAndLastNamesRequestMessage">
  <part name="firstName" type="xsd:string"/>
  <part name="lastName" type="xsd:string"/>
  <part name="address" type="typens:address"/>
</message>

<message name="GetAddressFromNameRequestMessage">
  <part name="name" type="xsd:string"/>
</message>

<message name="GetAddressFromNameResponseMessage">
  <part name="address" type="typens:address"/>
</message>

<portType name="AddressBookPT">
  <operation name="addEntry">
    <input name="AddEntryWholeNameRequest"
     message="tns:AddEntryWholeNameRequestMessage"/>
  </operation>
  <operation name="addEntry">
    <input name="AddEntryFirstAndLastNamesRequest"
     message="tns:AddEntryFirstAndLastNamesRequestMessage"/>
  </operation>
  <operation name="getAddressFromName">
    <input name="GetAddressFromNameRequest"
     message="tns:GetAddressFromNameRequestMessage"/>
    <output name="GetAddressFromNameResponse"
      message="tns:GetAddressFromNameResponseMessage"/>
  </operation>
</portType>

<binding name="SOAPHttpBinding" type="tns:AddressBookPT">
```

```
  <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryWholeNameRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="addEntry">
    <soap:operation soapAction=""/>
    <input name="AddEntryFirstAndLastNamesRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
  </operation>
  <operation name="getAddressFromName">
    <soap:operation soapAction=""/>
    <input name="GetAddressFromNameRequest">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output name="GetAddressFromNameResponse">
      <soap:body use="encoded"
                 namespace="http://www.ibm.com/namespace/wsif/samples/ab"
                 encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>

<binding name="JavaBinding" type="tns:AddressBookPT">
  <java:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="typens:address"
           formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
    <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
  </format:typeMapping>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="name address"
       methodType="instance"/>
    <input name="AddEntryWholeNameRequest"/>
  </operation>
  <operation name="addEntry">
    <java:operation
       methodName="addEntry"
       parameterOrder="firstName lastName address"
       methodType="instance"/>
    <input name="AddEntryFirstAndLastNamesRequest"/>
  </operation>
  <operation name="getAddressFromName">
    <java:operation
       methodName="getAddressFromName"
       parameterOrder="name"
       methodType="instance"
       returnPart="address"/>
    <input name="GetAddressFromNameRequest"/>
    <output name="GetAddressFromNameResponse"/>
  </operation>
</binding>

<binding name="EJBBinding" type="tns:AddressBookPT">
  <ejb:binding/>
```

```
    <format:typeMapping encoding="Java" style="Java">
      <format:typeMap typeName="typens:address"
             formatType="com.ibm.www.namespace.wsif.samples.ab.types.WSIFAddress"/>
      <format:typeMap typeName="xsd:string" formatType="java.lang.String"/>
    </format:typeMapping>
    <operation name="addEntry">
      <ejb:operation
         methodName="addEntry"
         parameterOrder="name address"
         interface="remote"/>
      <input name="AddEntryWholeNameRequest"/>
    </operation>
    <operation name="addEntry">
      <ejb:operation
         methodName="addEntry"
         parameterOrder="firstName lastName address"
         interface="remote"/>
      <input name="AddEntryFirstAndLastNamesRequest"/>
    </operation>
    <operation name="getAddressFromName">
      <ejb:operation
         methodName="getAddressFromName"
         parameterOrder="name"
         interface="remote"
         returnPart="address"/>
      <input name="GetAddressFromNameRequest"/>
      <output name="GetAddressFromNameResponse"/>
    </operation>
  </binding>
  <service name="AddressBookService">
    <port name="SOAPPort" binding="tns:SOAPHttpBinding">
      <soap:address
       location="http://localhost/wsif/samples/addressbook/soap/servlet/rpcrouter"/>
    </port>
    <port name="JavaPort" binding="tns:JavaBinding">
      <java:address className="services.addressbook.WSIFAddressBook"/>
    </port>
    <port name="EJBPort" binding="tns:EJBBinding">
      <ejb:address className="services.addressbook.ejb.AddressBookHome"
        jndiName="ejb/samples/wsif/AddressBook"
  classLoader="services.addressbook.ejb.AddressBook.ClassLoader"/>
    </port>
  </service>

</definitions>
```

Using complex types

WSIF supports user-defined complex types through the mapping of complex types to Java classes.

You specify this mapping manually or automatically as described in the following sections:

- Manual mapping of complex types.
- 5.0.2 + Automatic mapping of complex types.

Any calls to the WSIFService mapType and mapPackage methods used for manual mapping override any equivalent mapping information that is produced automatically. This override helps to maintain backwards compatibility, and also accommodates less standard mappings.

Manual mapping of complex types

The method to use when you create these mappings manually depends on the provider that is used. For the Java and EJB providers, the mappings are specified in the WSDL file in the binding element. The following example provides the syntax for specifying the mapping:

```
<binding .... >
    <ejb:binding|java:binding/>
      <format:typeMapping style="Java" encoding="Java"/>?
          <format:typeMap name="qname" formatType="nmtoken"/>*
      </format:typeMapping>
 ...
</binding>
```

In this example:
- A question mark ("?") means "optional" and an asterisk ("*") means "0 or more".
- The format:typeMap **name** attribute is a qualified name of a complex type or simple type used by one of the operations.
- The format:typeMap **formatType** attribute is the fully qualified class name for the Java class to which the element specified by **name** maps.

If you use the Apache SOAP provider then you specify the mapping of a complex type to a Java class in the client code through two methods on the org.apache.wsif.WSIFService interface:

```
public void mapType(QName elementType, Class javaType)
```

and

```
public void mapPackage(String namespaceURI, String packageName)
```

Use the **mapType** method to specify a mapping between an XML schema element and a Java class. The method takes a QName representing the complex type or simple type, and the corresponding Java class to which it maps.

Use the **mapPackage** method to specify a more general mapping between a namespace and a Java package. Any custom, complex or simple type whose namespace matches that of the mapping is mapped to a Java class in the corresponding package. The name of the actual class is derived from the name of the complex type using standard XML to Java naming conventions.

Automatic mapping of complex types 5.0.2 +

For complex types defined in the WSDL, where a generated bean is used to represent this type in Java, the Web Services Invocation Framework (WSIF) programming model requires that a call is made to the WSIFService.mapType() method. This call tells WSIF the package and class name of the bean representing the XML schema type that is identified with a QName. To make things easier, the WSIFService.mapPackage() method provides a mechanism to specify a wildcard version of this, where any class within a specified package is mapped to the namespace of a QName. This is a mechanism for manually mapping an XML schema type to a Java class and back again (one mapping entry provides a bidirectional mapping).

There are many ways to convert a QName representing an XML schema type name to a Java package name and class. To enable automatic type mapping, set the WSIF_FEATURE_AUTO_MAP_TYPES feature on the WSIFServiceFactory instance:

```
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
factory.setFeature(WSIFConstants.WSIF_FEATURE_AUTO_MAP_TYPES, new Boolean(true));
```

WSIF maps types by converting the URI part of the XML schema type <tt>QName</tt> to a package name, and converting the local part to a class name. WSIF does this mapping using the WSIFUtils methods <tt>getPackageNameFromNamespaceURI</tt> and <tt>getJavaClassNameFromXMLName</tt>.

Using the Java Naming and Directory Interface (JNDI)

This example task shows you how to use WSIF to bind a reference to a Web service, then look up the reference using JNDI.

You access a Web service through information provided in the WSDL document for the service. If you do not know where to find the WSDL document for the service, but you know that it has been registered in a UDDI registry, then you look it up in the registry. Java programs access Java objects and resources in a similar manner, but using a JNDI interface.

The following example shows how, using the Web Services Invocation Framework (WSIF), you can bind a reference to a Web service then look up the reference using JNDI.

Specifying the argument values for the Web service

The Web service is represented in WSIF by an instance of the org.apache.wsif.naming.WSIFServiceRef class. This simple Referenceable object has the following constructor:

```
public WSIFServiceRef(
        String WSDL,
        String sNS,
        String sName,
        String ptNS,
        String ptName)
{
    [...]
}
```

In this example

- *WSDL* is the location of the WSDL file containing the definition of the service.
- *sNS* is the full namespace for the service definition (you can specify `null` if only one service is defined in the WSDL file).
- *sName* is the local name for the service definition (you can specify `null` if only one service is defined in the WSDL file).
- *ptNS* is the full namespace for the port type within the service that you want to use (you can specify `null` if only one port type is available for the service).
- *ptName* is the local name for the port type (you can specify `null` if only one port type is available for the service).

For example, if the WSDL file for the Web service is available from the Web address `http://localhost/WSDL/Example.WSDL` and contains the following service and port type definitions:

```
  <definitions targetNamespace="http://hostname/namespace/example"
               xmlns:abc="http://hostname/namespace/abc"
[...]
    <portType name="ExamplePT">
      <operation name="exampleOp">
        <input name="exampleInput" message="tns:ExampleInputMsg"/>
      </operation>
    </portType>
[...]
    <service name="abc:ExampleService">
[...]
    </service>
[...]
  </definitions>
```

You can specify the following argument values for the WSIFServiceRef class:

- *WSDL* is `http://localhost/WSDL/Example.WSDL`
- *sNS* is `http://hostname/namespace/abc`
- *sName* is `ExampleService`
- *ptNS* is `http://hostname/namespace/example`
- *ptName* is `ExamplePT`

Binding the service using JNDI

To bind the service reference in the naming directory using JNDI, you can use the com.ibm.websphere.naming.JndiHelper class in WebSphere Application Server:

```
[...]
    import com.ibm.websphere.naming.JndiHelper;
    import org.apache.wsif.naming.*;
[...]
    try {
       Context startingContext = new InitialContext();
       WSIFServiceRef ref = new WSIFServiceRef("http://localhost/WSDL/Example.WSDL",
                                               "http://hostname/namespace/abc"
                                               "ExampleService",
                                               "http://hostname/namespace/example",
                                               "ExamplePT");
       JndiHelper.recursiveRebind(startingContext,
            "myContext/mySubContext/myServiceRef", ref);

    }
    catch (NamingException e) {
       // Handle  error.
    }
[...]
```

Looking up the service using JNDI

The following code fragment shows the lookup of a service using JNDI:

```
[...]
    try {
[...]
       InitialContext ic = new InitialContext();
       WSIFService myService =
        (WSIFService) ic.lookup("myContext/mySubContext/myServiceRef");
[...]
    }
    catch (NamingException e) {
       // Handle error.
    }
[...]
```

Passing SOAP messages with attachments using WSIF

The W3C SOAP Messages with Attachments document describes a standard way to associate a SOAP message with one or more attachments in their native format (for example GIF or JPEG) by using a multipart MIME structure for transport. It defines specific use of the "Multipart/Related" MIME media type, and rules for the use of URI references to entities bundled within the MIME package. It thereby outlines a technique for carrying a SOAP 1.1 message within a MIME multipart/related message in such a way that the SOAP processing rules for a standard SOAP message are not changed.

The Web Services Invocation Framework (WSIF) supports passing attachments in a MIME message using the SOAP provider. The attachment is a javax.activation.DataHandler object. The mime:multipartRelated, mime:part and mime:content tags are used to describe the attachment in the WSDL.

For more information, see the following topics:
- SOAP messages with attachments - Writing the WSDL extensions.
- SOAP messages with attachments - Passing attachments to WSI.
- SOAP messages with attachments - Working with types and type mappings.

The following scenarios are not supported:
- Using DIME.
- Passing in `javax.xml.transform.Source` and `javax.mail.internet.MimeMultipart`.
- Using the mime:mimeXml WSDL tag.
- Nesting a mime:multipartRelated tag inside a mime:part tag.
- Using types that extend `DataHandler`, `Image`, and so on.
- Using types that contain `DataHandler`, `Image`, and soon.
- Using Arrays or Vectors of `DataHandlers`, `Images`, and so on.
- Using multiple in/out or output attachments.

The MIME headers from the incoming message are not preserved for referenced attachments. The outgoing message contains new MIME headers for Content-Type, Content-Id and Content-Transfer-Encoding that are created by WSIF.

SOAP messages with attachments - Writing the WSDL extensions

The following example WSDL illustrates a simple operation that has one attachment called `attch`:

```
<binding name="MyBinding" type="tns:abc" >
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="MyOperation">
    <soap:operation soapAction=""/>
    <input>
      <mime:multipartRelated>
        <mime:part>
          <soap:body use="encoded" namespace="http://mynamespace"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
        </mime:part>
        <mime:part>
          <mime:content part="attch" type="text/html"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
  </operation>
</binding>
```

In this type of WSDL extension:

- There must be a `part` attribute (in this example `attch`) on the input message for the operation (in this example `MyOperation`). There can be other input parts to `MyOperation` that are not attachments.
- In the binding input there must either be a `<soap:body>` tag or a `<mime:multipartRelated>` tag, but not both.
- For MIME messages, the `<soap:body>` tag is inside a `<mime:part>` tag. There must only be one `<mime:part>` tag that contains a `<soap:body>` tag in the binding input and that must not contain a `<mime:content>` tag as well, because a content type of `text/xml` is assumed for the `<soap:body>` tag.
- There can be multiple attachments in a MIME message, each described by a `<mime:part>` tag.
- Each `<mime:part>` tag that does not contain a `<soap:body>` tag contains a `<mime:content>` tag that describes the attachment itself. The `type` attribute inside the `<mime:content>` tag is not checked or used by the Web Services Invocation Framework (WSIF). It is there to suggest to the application using WSIF what the attachment contains. Multiple `<mime:content>` tags inside a single `<mime:part>` tag means that the backend service expects a single attachment with a type specified by one of the `<mime:content>` tags inside that `<mime:part>` tag.
- The `parts="..."` attribute (optional) inside the `<soap:body>` tag is assumed to contain the names of all the MIME parts as well as the names of all the SOAP parts in the message.

SOAP messages with attachments - Passing attachments to WSIF

The following code fragment can invoke the service described by the example WSDL in the topic writing the WSDL extensions:

```
import javax.activation.DataHandler;
. . .
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
WSIFOperation op = service.getPort().createOperation("MyOperation");
WSIFMessage in = op.createInputMessage();
in.setObjectPart("attch",dh);
op.executeInputOnlyOperation(in);
```

The associated type mapping in the `DeploymentDescriptor.xml` file depends upon your SOAP server. For example if you use Tomcat with SOAP 2.3, then the `DeploymentDescriptor.xml` file contains the following type mapping:

```
<isd:mappings>
<isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:x="http://mynamespace"
 qname="x:datahandler"
 javaType="javax.activation.DataHandler"
 java2XMLClassName="org.apache.soap.encoding.soapenc.MimePartSerializer"
 xml2JavaClassName="org.apache.soap.encoding.soapenc.MimePartSerializer" />
</isd:mappings>
```

In this case, the backend service is invoked with the following signature:

```
public void MyOperation(DataHandler dh);
```

You can also use stubs to pass attachments into the Web Services Invocation Framework (WSIF):

```
DataHandler dh = new DataHandler(new FileDataSource("myimage.jpg"));
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService("my.wsdl",null,null,"http://mynamespace","abc");
MyInterface stub = (MyInterface)service.getStub(MyInterface.class);
stub.MyOperation(dh);
```

Attachments can also be returned from an operation, but at present only one attachment can be returned as the return parameter.

SOAP messages with attachments - Working with types and type mappings

By default, attachments are passed into the Web Services Invocation Framework (WSIF) as DataHandler objects. If the part on the message that is the DataHandler object maps to a `<mime:part>` tag in the WSDL, then WSIF automatically maps the fully qualified name of the WSDL type to the DataHandler class and sets up that type mapping with the SOAP provider.

In your WSDL, you might have defined a schema for the attachment (for instance as a `binary[]` type). WSIF silently ignores this mapping and treats the attachment as a DataHandler object, unless you explicitly issue a `mapType()` method. WSIF lets the SOAP provider set the MIME content type based on the type of the DataHandler object, instead of the `type` attribute specified for the `<mime:content>` tag in the WSDL.

Interacting with the J2EE container in WebSphere Application Server

Interaction with a container is limited to the following aspects:

- Using the application server administrative console to define Web services to WebSphere Application Server. This task is described in Using the Java Naming and Directory Interface (JNDI) and WSIF system management and administration. As part of the definition of a service, the administrator might define a “preferred port”.
- Using the Web Services Invocation Framework (WSIF) to make log and trace calls to the JRAS services in WebSphere Application Server, as described in Trace and logging for WSIF.
- Using WSIF providers to access Java 2 platform, Enterprise Edition (J2EE) services. For example using the EJB provider to access the Java Naming and Directory Interface (JNDI) and make calls to remote enterprise beans.
- Using WSIF to wrap the use of container services so that, when WSIF is run in an unmanaged (thin) environment, the operation can succeed.

Running WSIF as a client

The Web Services Invocation Framework (WSIF) runs in the WebSphere Application Server application client container, and in similar clients from other suppliers.

To simplify the process of launching client applications in the WebSphere Application Server application client, use the `launchClient` tool as described in Running application clients.

WSIF system management and administration

The Web Services Invocation Framework (WSIF) is provided as a stand-alone JAR file named `wsif.jar`. The JAR file contains the core WSIF classes, and the Java, EJB, SOAP over HTTP and SOAP over JMS providers. Additional providers are packaged as separate JAR files.

When you install WebSphere Application Server, the `wsif.jar` file is put on the WebSphere or Java Virtual Machine (JVM) class path.

WSIF requires no further configuration. WSIF is a thin abstraction layer between application code and the relevant invocation infrastructure.

For specific information on other aspects of managing your WSIF system, see the following topics:
- Maintaining the WSIF properties file
- Enabling security for WSIF
- Trace and logging for WSIF
- Troubleshooting the Web Services Invocation Framework
- WSIF (Web Services Invocation Framework) messages

Maintaining the WSIF properties file

The Web Services Invocation Framework (WSIF) properties are stored in the `wsif.jar` file, in a properties file named `wsif.properties`. This properties file is kept on the class path, so that WSIF can find it and the client administrator can use it to configure WSIF.

Here is a copy of the initial contents of the `wsif.properties` file. All the possible properties are listed and described.

```
# Two properties are used to override which WSIFProvider is selected when there
# exists multiple providers supporting the same namespace URI. These properties are:
#
#    wsif.provider.default.CLASSNAME=N
#    wsif.provider.uri.M.CLASSNAME=URI
#
# CLASSNAME is the WSIFProvider class name
# N is the number of following default wsif.provider.uri.M.CLASSNAME properties
# M is a number from 1 to N to uniquely identify each wsif.provider.uri.M.CLASSNAME
#   property key.
# For example the following two properties would override the default SOAP provider
# to be the Apache SOAP provider:
#
# wsif.provider.default.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=1
# wsif.provider.uri.1.org.apache.wsif.providers.soap.ApacheSOAP.WSIFDynamicProvider_ApacheSOAP=\
# http://schemas.xmlsoap.org/wsdl/soap/
#

# maximum number of milliseconds to wait for a response to a synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined on invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

Enabling security for WSIF

The Web Services Invocation Framework (WSIF) interacts with a security manager in the following ways:

- WSIF runs in the Java 2 platform, Enterprise Edition (J2EE) security context without modification.
- When WSIF is run under a J2EE container, port implementations can use the security context to pass on security tokens or credentials as necessary.
- WSIF implementations can automatically convert J2EE security context into appropriate context for onward services.

For WSIF to interact effectively with the WebSphere Application Server security manager, you must set the following permissions in the `server.policy` file:

Permission	Required by SOAP and Java portion?	Required by EJB portion?	Additional notes
FilePermission to load the WSDL	-	-	This permission is only required when a WSDL file is referred to using the file:/// protocol
RuntimePermission “getClassLoader” for the context class loader for the current thread	Yes	No	
RuntimePermission “accessDeclaredMembers”	Yes	Yes	This permission is required by both portions for handling enterprise beans
PropertyPermission for system properties	Yes (read and write access)	Yes (write access only)	This permission is required by SOAP and many others
NetPermission “specifyStreamHandler”	Yes	Yes	This permission must be in either the SOAP and Java portion, or the EJB portion, but it need not be in both.
SocketPermission “*host_name*”, “resolve”	No	Yes	Where *host_name* is your host name (for example `localhost`)
SocketPermission “*host_name*:*port_no*”, “connect”	Yes	Yes	Where *host_name* is your host name (for example `localhost`) and *port_no* is your port number (for example `9080`).

Troubleshooting the Web Services Invocation Framework

For information on resolving WebSphere-level problems, see Diagnosing and fixing problems.

To identify and resolve Web Services Invocation Framework (WSIF)-related problems, you can use the standard WebSphere Application Server trace and logging facilities. If you encounter a problem that you think might be related to WSIF, you can check for error messages in the WebSphere Application Server administrative console, and in the application server `stdout.log` file. You can also enable the application server debug trace to provide a detailed exception dump.

A list of the WSIF run-time system messages, with details of what each message means, is provided in Message reference for WSIF.

Here is a checklist of major WSIF activities, with advice on common problems associated with each activity:

Create service
: Handcrafted WSDL can cause numerous problems. To help ensure that your WSDL is valid, use a tool such as WebSphere Studio to create your service.

Define transport mechanism
: For the Java Messaging Service (JMS), check that you have set up the Java Naming and Directory Interface (JNDI) correctly, and created the necessary connection factories and queues.

  For SOAP, make sure that the deployment descriptor file `dds.xml` is correct - preferably by creating it using WebSphere Studio or similar tooling.

Create client - Java code
: Follow the correct format for creating a WSIF service, port, operation and message. For examples of correct code, see the Address Book Sample.

Compile code (client and service)
: Check that the build path against code is correct, and that it contains the correct levels of JAR files.

  Create a valid EAR file for your service in preparation for deployment to a Web server.

Deploy service
: When you install and deploy the service EAR file, check carefully any messages given when the service is deployed.

Server setup and start
: Make sure that the WebSphere Application Server `server.policy` file (in the `/properties` directory) has the correct security settings. For more information, see Enabling security for WSIF.

WSIF setup
: Check that the `wsif.properties` file is correctly set up. For more information, see Maintaining the WSIF properties file.

Run client
: Either check that you have defined the class path correctly to include references to your client classes, WSIF JAR files and any other necessary JAR files, or (preferably) run your client using the WebSphere Application Server launchClient tool.

Here is a list of common errors, and information on their probable causes:

- **"No class definition" errors received when running client code.**

  This problem usually indicates an error in the class path setup. Check that the relevant JAR files are included.
- **"Cannot find WSDL" error.**

  Some likely causes are:
  - The application server is not running.
  - The server location and port number in the WSDL are not correct.
  - The WSDL is badly formed (check the error messages in the application server `stdout.log` file).
  - The application server has not been restarted since the service was installed.

  You might also try the following checks:
  - Can you load the WSDL into your Web browser from the location specified in the error message?

– Can you load the corresponding WSDL binding files into your Web browser?

- **Your Web service EAR file does not install correctly onto the application server.**

  It is likely that the EAR file is badly formed. Verify the installation by completing the following steps:
  – For an EJB binding, run the WebSphere Application Server tool `\bin\dumpnamespace`. This tool lists the current contents of the JNDI directory.
  – For a SOAP over HTTP binding, open the `http://pathToServer/WebServiceName/admin/list.jsp` page (if you have the SOAP administration pages installed). This page lists all currently installed Web services.
  – For a SOAP over JMS binding, complete the following checks:
    - Check that the queue manager is running.
    - Check that the necessary queues are defined.
    - Check the JNDI setup.
    - Use the “display context” option in the `jmsadmin` tool to list the current JNDI definitions.
    - Check that the Remote Procedure Call (RPC) router is running.
- **There is a permissions problem or security error.**

  Check that the WebSphere Application Server `server.policy` file (in the `/properties` directory) has the correct security settings. For more information, see Enabling security for WSIF.
- **Using WSIF with multiple clients causes a SOAP parsing error.**

  Before you deploy a Web service to WebSphere Application Server, you must decide on the scope of the Web service. The deployment descriptor file `dds.xml` for the Web service includes the following line:

```
<isd:provider type="java" scope="Application" ......
```

  You can set the `Scope` attribute to `Application` or `Session`. The default setting is `Application`, and this value is correct if each request to the Web service does not require objects to be maintained for longer than a single instance. If `Scope` is set to `Application` the objects are not available to another request during the execution of the single instance, and they are released on completion. If your Web service needs objects to be maintained for multiple requests, and to be unique within each request, you must set the scope to `Session`. If `Scope` is set to `Session`, the objects are not available to another request during the life of the session, and they are released on completion of the session. If scope is set to `Application` instead of `Session`, you might get the following SOAP error:

```
SOAPException: SOAP-ENV:ClientParsing error, response was:
FWK005 parse may not be called while parsing.;
nested exception is:

[SOAPException: faultCode=SOAP-ENV:Client; msg=Parsing error, response was:

FWK005 parse may not be called while parsing.;
       targetException=org.xml.sax.SAXException:
FWK005 parse may not be called while parsing.]
```

Trace and logging for WSIF

If you want to enable trace for the Web Services Invocation Framework (WSIF) API within WebSphere Application Server, and have trace, stdout and stderr for the application server written to a well-known location, see Setting up component trace (CTRACE).

WSIF offers trace points at the opening and closing of ports, the invocation of services, and the responses from services.

To trace the WSIF API, you need to specify the following trace string:

```
wsif=all=enabled
```

WSIF also includes a `SimpleLog` utility through which you can run trace when using WSIF outside of WebSphere Application Server. To enable this utility, complete the following steps:

1. Create a file named `commons-logging.properties` with the following contents:

```
org.apache.commons.logging.LogFactory=org.apache.commons.logging.impl.LogFactoryImpl
org.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
```

2. Create a file named `simplelog.properties` with the following contents:

```
org.apache.commons.logging.simplelog.defaultlog=trace
org.apache.commons.logging.simplelog.showShortLogname=true
org.apache.commons.logging.simplelog.showdatetime=true
```

3. Put both these files, and the `commons-logging.jar` file, on the class path.

The `SimpleLog` uitility writes trace to the `System.err` file.

WSIF (Web Services Invocation Framework) messages

This topic contains a list of the WSIF run-time system messages, with details of what each message means.

WebSphere system messages are logged from a variety of sources, including application server components and applications. Messages logged by application server components and associated IBM products start with a unique message identifier that indicates the component or application that issued the message.

WSIF0001E: An extension registry was not found for the element type "{0}"

Explanation: Parameters: {0} element type. No extension registry was found for the element type specified.

User Response: Add the appropriate extension registry to the port factory in your code.

WSIF0002E: A failure occurred in loading WSDL from "{0}"

Explanation: Parameters: {0} location of the WSDL file. The WSDL file could not be found at the location specified or did not parse correctly

User Response: Check that the location of the WSDL file is correct. Check that any network connections required are available. Check that the WSDL file contains valid WSDL.

WSIF0003W: An error occurred finding pluggable providers: {0}

Explanation: Parameters: {0} specific details about the error. There was a problem locating a WSIF pluggable provider using the J2SE 1.3 JAR file extensions to support service providers architecture. The WSIF trace file will contain the full exception details.

User Response: Verify that a META-INF/services/org.apache.wsif.spi.WSIFProvider file exists in a provider jar, that each class referenced in the META-INF file exists in the class path, and that each class implements org.apache.wsif.spi.WSIFProvider. The class in error will be ignored and WSIF will continue locating other pluggable providers.

WSIF0004E: WSDL contains an operation type "{0}" which is not supported for "{1}"

Explanation: Parameters: {0} name of the operation type specified. {1} name of the portType for the operation. An operation type which is not supported has been specified in the WSDL.

User Response: Remove any operations of the unsupported type from the WSDL. If the operation is required then make sure all messages have been correctly specified for the operation.

WSIF0005E: An error occurred when invoking the method "{1}" . ("{0}")

Explanation: Parameters: {0} name of communication type. For example EJB or Apache SOAP. {1} name of the method that failed. An error was encountered when invoking a method on the Web service using the communication shown in brackets.

User Response: Check that the method exists on the Web service and that the correct parts have been added to the operation as described in the WSDL. Network problems might be a cause if the method is remote and so check any required connections.

WSIF0006W: Multiple WSIFProvider found supporting the same namespace URI "{0}" . Found ("{1}")

Explanation: Parameters: {0} the namespace URI. {1} a list of the WSIFProvider found.. There are multiple org.apache.wsif.spi.WSIFProvider classes in the service provider path that support the same namespace URI.

User Response: A following WSIF0007I message will be issued notifying which WSIPFProvider will be used. Which WSIFProvider is chosen is based on settings in the wsif.properties file, or if not defined in the properties, the last WSIFProvider found will be used. See the wsif.properties file for more details on how to define which provider should be used to support a namespace URI.

WSIF0007I: Using WSIFProvider "{0}" for namespaceURI "{1}"

Explanation: Parameters: {0} the classname of the WSIFProvider being used. {1} the namespaceURI the provider will be used to support.. Either a previous WSIF0006W message has been issued or the SetDynamicWSIFProvider method has been used to override the provider used to support a namespaceURI.

User Response: None. See also WSIF0006W.

WSIF0008W: WSIFDefaultCorrelationService removing correlator due to timeout. ID:"{0}"

Explanation: Parameters: {0} the ID of the correlator being removed from the correlation service. A stored correlator is being removed from the correlation service due to its timeout expiring.

User Response: Determine why no response has been received for the asynchronous request within the timeout period. The wsif.asyncrequest.timeout property of the wsif.properties file defines the length of the timeout period.

WSIF0009I: Using correlation service - "{0}"

Explanation: Parameters: {0} the name of the correlation service being used. This identifies the name of the correlation service that will be used to process asynchronous requests.

User Response: None. If a correlation service other than the default WSIF supplied one is required, ensure that it is correctly registered in the JNDI java:comp/wsif/WSIFCorrelationService namespace.

WSIF0010E: Exception thrown while processing asynchronous response - "{0}"

Explanation: Parameters: {0} the error message string of the exception. While processing the response from an executeRequestResponseAsync call an exception was thrown.

User Response: Use the exception error message string to determine the cause of the error. The WSIF trace will have more details on the error including the exception stack trace.

WSIF0011I: Preferred port “{0}” was not available

Explanation: Parameters: {0} the user’s preferred port. The preferred port set by the user on org.apache.wsif.WSIFService is not available

User Response: None unless this message appears for long periods of time in which case the user might want to pick a different port as their preferred port.

WSIF API

The Web Services Invocation Framework (WSIF) API supports the invocation of services defined in WSDL. WSIF is intended for use in both WSIF clients and Web service intermediaries.

The WSIF API is driven by the abstract service description in WSDL; it is completely independent of the actual binding used. This independence makes the API more natural to work with because it uses WSDL terms to refer to message parts, operations, and so on.

The WSIF API was designed for the WSDL usage model: Pick a port that supports the port type needed, then invoke the operation by providing the necessary abstract input message consisting of the required parts, without worrying about how the message is mapped to a specific binding protocol.

Other Web service APIs, for example SOAP APIs, are not designed on WSDL, but for a specific binding protocol with its associated syntax; for example, target URIs and encoding styles.

The WSIF API main interfaces are described in the following topics:
- Creating a message for sending to a port (the WSIFMessage interface).
- WSIF API reference: Finding a port factory or service (the WSIFService interface and the WSIFServiceFactory class).
- WSIF API reference: Using ports (the WSIFPort interface and the WSIFOperation interface).

Note: You must ensure that your application uses only one thread to call WSIF.

For additional technical details of the WSIF API, see the WSIF Javadoc.

WSIF API reference: Creating a message for sending to a port

For message management (that is, message construction and parsing) the underlying API is modeled on WSDL semantics. There is a simple and direct mapping from the WSDL model to the Web Services Invocation Framework (WSIF) classes.

In WSDL, a message describes the abstract type of the input or output to an operation. The corresponding WSIF class is WSIFMessage, which represents in memory the actual input or output of an operation. A WSIFMessage class is a container for a set of named parts. The WSIFMessage interface separates the actual representation of the data from the abstract type defined by WSDL. WSDL defines messages as XML schema types. There are two natural ways to represent a WSDL message in a run-time environment:
- The generated Java class, based on a WSDL to Java mapping such as that provided by a Java API for XML-based remote procedure call (JAX-RPC).
- The XML representation of the data, for example using SOAP Encoding.

Each option offers benefits in different scenarios. The Java class is the natural approach when WSIF is used in a standard Java client. However, in other scenarios where WSIF is used in an intermediary, it might be more efficient to keep a WSDL message in the SOAP encoded format.

The style used to define messages must be consistent within the message, so all the parts in one message must be consistent. A string - `getRepresentationStyle()` - always returns `null`. This indicates that parts on this WSIFMessage class are represented as Java objects.

You add parts to a WSIFMessage class with the setObjectPart or set*Type*Part methods. Each part is named. Part names within a message are unique. If you set a part more than once, the last setting is the one that is used.

You retrieve parts by name from a WSIFMessage class with the getObjectPart or get*Type*Part methods. If the named part does not exist, the method returns a WSIFException exception.

You can use Iterators to retrieve parts from the message through the getParts() and getPartNames() methods.

The order in which you set the parts is not important, but the message implementation might be more efficient if the parts are set in the parameter order specified by WSDL.

WSIFMessage classes are cloneable and serializable. If the parts set are not cloneable, the implementation can try to clone them using serialization. If the parts are not serializable either, then a CloneNotSupportedException exception is thrown if cloning is attempted.

WSIFMessage classes can be sent between Java Virtual Machines (JVMs).

In addition to the containing parts, a WSIFMessage class also has a message name. This is required for operation overloading, which is supported by WSDL and WSIF.

Here is the Javadoc for the WSIFMessage interface.

WSIF API reference: Finding a port factory or service

To find a port you use the WSIFService interface, which is a factory for ports.

The port factory models and supports the WSDL approach in which a service is available on one or more ports. The factory hides the implementation of the port from the user. The Web Services Invocation Framework (WSIF) supports dynamic ports that are based on a particular protocol and transport, and configured using the WSDL at run-time. For example, the dynamic SOAP port can invoke any SOAP service based on the WSDL description of that service. Using this service you can hide and modify the set of available ports at run-time.

Here is the WSIFService interface.

To find a service from a WSDL document at a Web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

WSIFService interface

The WSIFService interface is responsible for generating an instance of the WSIFOperation interface to use for a particular invocation of a service operation.

The Web Services Invocation Framework (WSIF) service stores a list of providers that can each generate a WSIF operation for a particular WSDL binding. This service looks up providers by the provider type. For example the service knows about one provider that handles SOAP ports and other providers that handle Java ports that you define. In a managed environment, the container can configure the WSIFService interface.

Here is the Javadoc for the WSIFService interface.

A WSIFService implementation can choose a preferred port based on a number of criteria. The WSIFService implementation can set the preferred port, or it can be set by calling the setPreferredPort method.

The getPort method returns an instance of the WSIFPort class that is used to invoke a service on the port. Variants of the getPort method are used to define the characteristics of the port to be created:

- the getPort method with no arguments returns the preferred port.
- the getPort method with a string argument returns the port named by the string containing the WSDL identifier for the selected port.

The return value is `null` if the port name is not valid.

If a port is chosen (either by the WSIFService implementation, or by the setPreferredPort method), then the WSIFService implementation validates that the relevant provider exists and is configured. If the provider fails this validation check, the WSIFService interface chooses any other port for which a provider is defined. For example, if the preferred port is SOAP over JMS but the JMS libraries are not available, then WSIF chooses another port. If no preferred port is set, or the preferred port is not available, the WSIF implementation chooses the first available port listed in the WSDL.

The getAvailablePortNames() method returns, as an iteration of strings, the list of WSDL port names filtered by the set of available providers.

The getDefinition() method returns the WSDL definition for the service. If the WSDL definition is not available, this method returns `null`.

WSIFServiceFactory class

To find a service from a WSDL document at a Web address, or from a code-generated code base, you can use the WSIFServiceFactory class.

Note: When you create a WSIFService interface from a WSIFServiceFactory class, you can specify a ClassLoader object to use in locating the WSDL file. You need to specify this object when the WSDL file is in a JAR file. In such a case, specify the location of the WSDL file relative to the root of the JAR file, using forward slashes (/) with the preceding slash removed.

For example:

```
com/myCompany/wsdl/MyWSDLFile.wsdl
```

rather than

```
/com/myCompany/wsdl/MyWSDLFile.wsdl
```

Here is the Javadoc for the WSIFServiceFactory class.

The WSIFServiceFactory class returns `null` if no service is found with that identifier.

WSIF API reference: Using ports

A WSIFPort interface handles the details of invoking an operation. The port provides access to the actual implementation of the service.

A WSDL document can provide many different WSDL bindings, and these bindings can drive multiple ports. The client can choose a port, the service stub can choose a port, or the Web Services Invocation Framework (WSIF) can choose a default port.

The port offers an interface to retrieve an Operation object. A WSIFOperation interface offers the ability to execute the given operation.

If the port is serialized and deserialized at a later time, then WSIF ensures that the client provides the correct information to the server to identify the instance. If the server instance is no longer available, then it is up to the server to decide whether to throw a fault or provide a new instance. That behavior can depend on the type of service.

For example, for an enterprise bean the WSIFPort interface stores the EJB Home, and uses it to select the bean before each invocation. It is the responsibility of the client to serialize or maintain the port instance if it wants instance support. The client must create a new operation and messages for each invocation.

Here is the WSIFPort interface.

Here is the WSIFOperation interface.

WSIFPort interface

The port implements a factory method for the WSIFOperation interface.

Here is the Javadoc for the WSIFPort interface.

The createOperation(String) method returns a new instance of a WSIFOperation object. If the operationName value is not valid or the operation is overloaded, then the method throws an exception.

The createOperation(String, String, String) method supports overloaded WSDL operations. You can overload based on the input parameters, but not on the output parameters.

It is the duty of the client to call the close method when a port is no longer in use. In many cases, where the transport is sessionless, like HTTP, this has no effect. However, if the port is using a session-based protocol such as MQSeries, Java Messaging Service (JMS), or External Call Interface (ECI), this supports the port in caching an open connection to the server and then closing it as required. Responsibly-written applications will call the close method if appropriate.

WSIFOperation interface

You use the WSIFOperation interface to invoke a service based on a particular binding.

The WSIFOperation interface is the run-time representation of an operation. This interface provides methods to create input, output, and fault messages, and to invoke the operation.

Here is the Javadoc for the WSIFOperation interface.

createInputMessage, createOutputMessage and createFaultMessage
These are factory methods to create the messages required by the invocation methods. All invocation methods require an input message.

executeRequestResponseOperation
This method invokes "In Out" operations.

executeInputOnlyOperation
This method invokes "In only" operations.

executeRequestResponseOperation
If this method is used for invocation, then an output and a fault message are instantiated and passed on the call to the method. If the method returns `true`, then the output message contains the response message. If the message returns `false`, then a fault occurred and is returned in the fault message.

executeRequestResponseAsync
This method allows "In Out" operations to be invoked with the reply handled using an alternate thread. Use of this method is discussed further in WSIFOperation - Asynchronous interactions.

setContext and getContext
Use of these methods is discussed in WSIFOperation - Context.

All of the **execute*Nnnn*** methods fail with an exception if there is an error in processing the request in the WSIF provider.

Setting the timeouts for synchronous and asynchronous operations is discussed in WSIFOperation - Synchronous and asynchronous timeouts.

WSIFOperation - Context: Although WSDL does not define context, a number of uses of the Web Services Invocation Framework (WSIF) require the ability to pass context to the port that is invoking the service. For example, a SOAP over HTTP port might require an HTTP user name and password. This information is specific to the invocation, but is not a parameter of the service. In general, context is defined as a set of name-value pairs. However, because Web services tend to define the types of data using XML schema types, WSIF represents the name-value pairs of the context using the same representation that WSIFMessage classes use; that is a set of named parts, each of which equates to an instance of an XML schema type.

You use the WSIFOperation interface setContext and getContext methods to pass context information to the binding. The port implementation can use this context, for example to update a SOAP header. There is no definition of how a port can utilize the context.

The parameter of the setContext and getContext methods is a WSIFMessage interface, and this interface has named parts defining the context information. The WSIFConstants class defines constants for the part names that can be set in a context WSIFMessage interface.

The following code fragment shows how to set the user name and password for HTTP basic authentication:

```
// set a basic authentication header
   WSIFMessage headers = new WSIFDefaultMessage();
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_USER, "user name" );
   headers.setObjectPart( WSIFConstants.CONTEXT_HTTP_PSWD, "password" );
   operation.setContext( headers );
```

The WSIFOperation interface ignores context parts that it does not support. For example, the previous code is ignored by the WSIF Java provider.

The WSIFConstants class includes the following constants that can be used for context part names:
- CONTEXT_HTTP_USER
- CONTEXT_HTTP_PSWD
- CONTEXT_SOAP_HEADERS

The HTTP header values are expected to be of type String, and the SOAP header value is expected to be of type java.util.List, which should contain entries of type org.w3c.dom.Element.

WSIFOperation - Asynchronous interactions reference: The Web Services Invocation Framework (WSIF) supports asynchronous operation. In this mode of operation, the client puts the request message as part of one transaction, and carries on with the thread of execution. The response message is then handled by a different thread, with a separate transaction. The SOAP over JMS and native JMS providers are the only WSIF providers that currently support asynchronous operation.

The WSIFPort class uses the supportsAsync method to test if asynchronous operation is supported.

An asynchronous operation is initiated with the WSIFOperation interface executeRequestResponseAsync method. This method lets a Remote Procedure Call (RPC) method be invoked asynchronously. The method returns before the operation is completed, and the thread of execution continues.

The response to the asynchronous request is processed by the WSIFOperation interface fireAsyncResponse or processAsyncResponse methods.

To initiate the request, there are two forms of the executeRequestResponseAsync method:

```
public WSIFCorrelationId executeRequestResponseAsync
                         (WSIFMessage input, WSIFResponseHandler handler)
```

and

```
public WSIFCorrelationId executeRequestResponseAsync (WSIFMessage input)
```

executeRequestResponseAsync(WSIFMessage input, WSIFResponseHandler handler)
: This method takes an input message and a WSIFResponseHandler handler. The handler is invoked on another thread when the operation completes. When using this method the client listener calls the fireAsyncResponse method, which then calls the WSIFResponseHandler interface executeAsyncResponse method. Here is the Javadoc for the WSIFResponseHandler interface.

executeRequestResponseAsync(WSIFMessage input)
: This method only takes an input message, and does not use a response handler. The client listener processes the response by calling the WSIFOperation interface processAsyncResponse method. This process updates the WSIFMessage output and fault messages with the result of the request.

WSIF supports correlation between the asynchronous request and response. When the request is sent, the WSIFOperation object is serialized into the WSIFCorrelationService object. The executeRequestResponseAsync methods return a WSIFCorrelationId object which identifies the serialized WSIFOperation object. The client listener can use this to match a response to a particular request.

The correlation service is located with the WSIFCorrelationServiceLocator class getCorrelationService() method in the org.apache.wsif.utils package.

In a managed container a default correlation service is defined in the default Java Naming and Directory Interface (JNDI) namespace using the name: java:comp/wsif/WSIFCorrelationService. If this correlation service is not available, then WSIF uses the WSIFDefaultCorrelationService.

Here is the Javadoc for the WSIFCorrelationService interface.

and this is the correlator ID:

```
public interface WSIFCorrelator extends Serializable {
   public String getCorrelationId();
}
```

The client must implement its own response message listener or message data base so that it can recognize the arrival of response messages. This client implementation manages the correlation of the response message to the request and call of one of the asynchronous response processing methods. As an example of the requirement for a client listener, the following code fragment shows what can be in the onMessage method of a Java Messaging Service (JMS) listener:

```
public void onMessage(Message msg) {
     WSIFCorrelationService cs = WSIFCorrelationServiceLocator.getCorrelationService();
           WSIFCorrelationId cid = new JmsCorrelationId( msg.getJMSCorrelationID() );
           WSIFOperation op = cs.get( cid );
           op.fireAsyncResponse( msg );
}
```

WSIFOperation - Synchronous and asynchronous timeouts reference:

When you use the Web Services Invocation Framework (WSIF) with the Java Messaging Service (JMS) you can set timeouts for synchronous and asynchronous operations.

Default values for these timeouts are defined in the WSIF properties file:

```
# maximum number of milliseconds to wait for response to synchronous request.
# Default value if not defined is to wait forever.
wsif.syncrequest.timeout=10000

# maximum number of seconds to wait for a response to an async request.
# if not defined on invalid defaults to no timeout
wsif.asyncrequest.timeout=60
```

If you use these default values, a synchronous request (such as a WSIFOperation interface executeRequestResponseOperation method call) times out after ten seconds, and an asynchronous request (such as a WSIFOperation interface executeRequestResponseAsync method call) times out after sixty seconds.

Note:

The code that processes both of these timeout values uses milliseconds as its unit of time. The WSIFProperties class getAsyncTimeout method multiplies the wsif.asyncrequest.timeout value by 1000, to convert the value from seconds to milliseconds.

You can override these default values for a given request by setting a JMS property on the operation request with the `<jms:property>` and `<jms:propertyValue>` WSDL elements. Set the name of the property to be the name of the timeout from the WSIF properties file.

The following example sets synchronous requests to time out after two minutes (120 seconds):

```
<jms:propertyValue name="wsif.syncrequest.timeout" type="xsd:string" value="120000"/>
```

and the following example disables asynchronous timeouts (a value of zero means wait forever):

```
<jms:propertyValue name="wsif.asyncrequest.timeout" type="xsd:string" value="0"/>
```

When an asynchronous timeout expires, no listener or message data base waiting for the response is notified. The asynchronous timeout is only used to tell the correlation service that the stored WSIFOperation can be deleted.

Chapter 9. IBM WebSphere UDDI Registry

Welcome to the IBM WebSphere UDDI Registry.

Use the table of contents (on the left and below)to view the various topics for a specific product or technology. Select the topic you are interested in to either open documentation locally or find information about how to locate documentation.

- Terminology
- Definitions
- Overview of UDDI Registries
- Installing the UDDI Registry Component
- Use of a remote DB2 Database
- Reinstalling the UDDI Registry Component
- Removing the UDDI Registry application from a deployment manager cell
- Removing the UDDI Registry application from a single appserver
- Configuring the UDDI Registry
- Administering the UDDI Registry
- The UDDI user console
- 5.0.2 + 5.0.2 + Custom Taxonomy Support in the UDDI Registry
- The SOAP Application Programming Interface
- The application programming interface
- The EJB Interface
- UDDI4J
- Problem determination
- Messages
- Samples
- Installation Verification Program (IVP)
- Reporting Problems with the IBM WebSphere UDDI Registry
- Feedback

UDDI Registry terminology

The directory location of the WebSphere Application Server is referred to as **<AppServer-install-dir>** and the directory location of the WebSphere Deployment manager as **<DeploymentManager-install-dir>**. The default locations are:

Windows

<AppServer-install-dir>
C:\Progra~1\WebSphere\AppServer\

<DeploymentManager-install-dir>
C:\Progra~1\WebSphere\DeploymentManager\

5.0.1 + **Linux/Solaris/HP Platforms**

<AppServer-install-dir>
/opt/WebSphere/AppServer/

<DeploymentManager-install-dir>
/opt/WebSphere/DeploymentManager/

AIX Platform

<AppServer-install-dir>
/usr/WebSphere/AppServer/

<DeploymentManager-install-dir>
/usr/WebSphere/DeploymentManager/

z/OS Platform
<AppServer-install-dir>
/WebSphere390/V5R0M0/AppServer/
<DeploymentManager-install-dir>
/WebSphere390/V5R0M0/DeploymentManager/

UDDI Registry definitions

bindingTemplate
Technical information about a service entry point and construction specifications.

businessEntity
Information about the party who publishes information about a family of services.

businessService
Descriptive information about a particular service.

publisherAssertion
Information about a relationship between two parties, asserted by one or both.

tModel
Short for technical model.

A tModel is a data structure representing a reusable concept, such as a Web service type, a protocol used by Web services, or a category system.

tModel keys within a service description are a technical "fingerprint" that you can use to trace the compatibility origins of a given service. They provide a common point of reference that allows you to identify compatible services.

tModels are used to establish the existence of a variety of concepts and to point to their technical definitions. tModels that represent value sets such as category, identifier, and relationship systems are used to provide additional data to the UDDI core entities to facilitate discovery along a number of dimensions. This additional data is captured in keyedReferences that reside in category Bags, identifierBags, or publisherAssertions. The tModelKey attributes in these keyedReferences refer to the value set that relates to the concept or namespace being represented. The keyValues contain the actual values from that value set. In some cases keyNames are significant, such as for describing relationships and when using the general keywords value set. In all other cases, however, keyNames are used to provide a human readable version of what is in the keyValue.

An overview of IBM UDDI Registries

The Universal Description, Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services. The term 'Web service' describes specific business functionality exposed by a company, usually through an Internet connection, to allow another company, or its subsidiaries, or software program to use the service.

Universal Business Registries (IBM UBR)

The IBM Universal Business Registry is one of a group of Web-based registries that expose information about a business or other entity and its technical interfaces (or

APIs). These registries are run by multiple Operator Sites, and can be used by anyone who wants to make information available about one or more businesses or entities, as well as anyone who wants to find that information. Although there are Universal Business Registries (sometimes referred to as ’public UDDI registries’) hosted worldwide, including one hosted by IBM, enterprises may wish to host their own internal registries behind their firewall to better manage their internal implementation of Web services.

For more detailed information about UDDI in general visit *http://www.uddi.org*

IBM WebSphere UDDI Registry

The IBM WebSphere UDDI Registry is a directory for Web services that is implemented using the UDDI specifications. In contrast with the IBM UBR, this component of WebSphere Network Deployment is a product offering for companies or industries to implement.

A critical component of IBM’s dynamic e-business infrastructure, IBM WebSphere UDDI Registry solves the problem of discovery of technical components for an enterprise and its partners by:
- Providing control, flexibility and confidentiality so that an enterprise can protect its e-business investments
- Increasing efficiency by making it easier to identify technical assets
- Leveraging existing infrastructures

For example, the IBM WebSphere UDDI Registry could be used in the following way within a large enterprise:

A company has a legacy application that provides telephone numbers and Human Resources (HR) information of employees. This is turned into a Web service and published to the registry. A developer in the same company needs to write an application for a procurement function that also needs to provide HR information to the supplier. The application should allow the supplier to have access to the employee account codes once the employee provides his name or serial number. Before Web Services, the developer had three choices:
1. Would not have known about the similar application
2. Knew about it but could not reuse due to technical barriers
3. Knew about it and reused only after significant time and negotiation

With UDDI, the developer can search for the ″web service″ and reuse the existing technical component in their new application for the supplier in a matter of minutes. The developer saves time and gets the application up and running sooner than they would have otherwise -- increasing efficiency and saving the company time and money. The IBM WebSphere UDDI Registry is the first version 2 standard-compliant UDDI registry for private enterprise work. The IBM WebSphere UDDI Registry:
- Supports the public UDDI V2.0 standard
- Leverages the proven, reliable WebSphere Application Server technology
- Uses a relational database, such as DB2, for its persistent store.

Installing and setting up a UDDI Registry

If you wish to use the UDDI User Console using Internet Explorer as your Web browser, and using SSL, you must use Internet Explorer V5.5 with SP2 and security fix Q321232 (which must be applied in that order), or later.

Choice of database product to be used as the persistence store

Cloudscape Restriction

Cloudscape Network Server Version 5.1 requires a WebSphere Version 5 datasource to utilize the multiple connection features. As IBM WebSphere UDDI Registry uses a WebSphere Version 4 datasource, this precludes other connections to the Cloudscape database when the UDDI Registry application is in the started state.

The UDDI Registry application can use either DB2 or Cloudscape as the persistence store for the registry data. However, in a z/OS environment, you should use DB2. If you use Cloudscape, you must restrict the WLM policy to the server to allow only one servant process.

Steps for this task

You are given the option to install the UDDI Registry as part of the IBM WebSphere Application Server for z/OS ISPF Customization Dialog. See *WebSphere Application Server for z/OS V5 Installation and Customization* for more information on how to install the UDDI Registry. The latest version of this publication is available on the product library page at URLhttp://www.ibm.com/software/webservers/appserv/zos_os390/library.html

In most cases you will probably choose option 1, and install the UDDI Registry into a deployment manager cell, but you might find that option 2, to install the UDDI Registry into a standalone application server, is useful for development or test purposes.

Note:

1. Several WebSphere commands are used during the following procedures, some of which must execute on the DeploymentManager and some of which must execute on the target Application server. The instructions distinguish which is appropriate for each command. The WebSphere commands are in the bin subdirectory of the appropriate WebSphere install tree. To ensure correct operation of these commands, do one of the following:
   - Ensure that the appropriate bin subdirectory is in your path prior to executing the command
   - Change directory to the appropriate bin subdirectory
   - Fully qualify the path to the commands
2. It is also recommended that you execute setupCmdline.bat (on Windows) or ./setupCmdline.sh (on Unix platforms) prior to executing any WebSphere commands.

The following table lists the UDDI Registry files, and the locations into which they are placed by the installation. The Location column shows the subdirectory under the WebSphere DeploymentManager install directory. For example, if you had installed IBM WebSphere Application Server with Network Deployment option onto a machine running Windows, and had used the default directory, then a

location of installableApps would mean that the file had been placed into the `C:\Progra~1\WebSphere\DeploymentManager\installableApps` directory. For Windows platforms, read the "/" directory separator in the location column as a "\" directory separation character.

Files	Purpose	Location
uddi.ear	The UDDI Registry application itself, which is packaged and runs as an enterprise application	installableApps
uddi.properties	Provides configuration properties for the UDDI Registry application	properties
uddiresourcebundles.jar	Contains system messages for the UDDI Registry application	lib
uddicloudscapeuserfunc.jar	Contains functions that are used by Cloudscape if the Cloudscape database is used with the UDDI Registry	lib
setupuddi.jacl	Administrative script to create a JDBC driver and datasource for the UDDI Registry, and to install the UDDI Registry application in a DeploymentManager Cell	UDDIReg/scripts
setupuddimessages.jar	Contains setup and install messages for the UDDI Registry application	lib
removeuddi.jacl	Administrative script to undo the effects of setupuddi.jacl	UDDIReg/scripts
appserverremoveuddi.jacl	Administrative script to undo the effects of appserversetupuddi.jacl	UDDIReg/scripts
appserversetupuddi.jacl	Administrative script to create a JDBC driver and datasource for the UDDI Registry, and to install the UDDI Registry application in a single, stand-alone, application server	UDDIReg/scripts
SetupDB2UDDI.jar	The 'UDDI DB2 Setup Wizard', to create and pre-load the UDDI Registry database if DB2 is to be used as the persistence store	UDDIReg/scripts
UDDI20 (directory)	Cloudscape directory containing the UDDI Registry tables and pre-loaded data	bin
uddiejbclient.jar	Class library for use when writing an EJB client to access the UDDI Registry	UDDIReg/ejb
Various javadoc files	JAVADOC to describe the EJB interface to the UDDI Registry	UDDIReg/ejb/javadoc
UDDITaxonomyTools.jar	Provides tools for supporting custom taxonomies with the UDDI Registry	UDDIReg/scripts
CustomTaxonomy.properties	Provides configuration properties to be used the the UDDITaxonomyTools	UDDIReg/scripts
UDDIUtilityTools.jar	Provides support for import/export of UDDI entities	UDDIReg/scripts
UDDIUtilityTools.properties	Provides configuration properties for the UDDI Utility Tools	UDDIReg/scripts

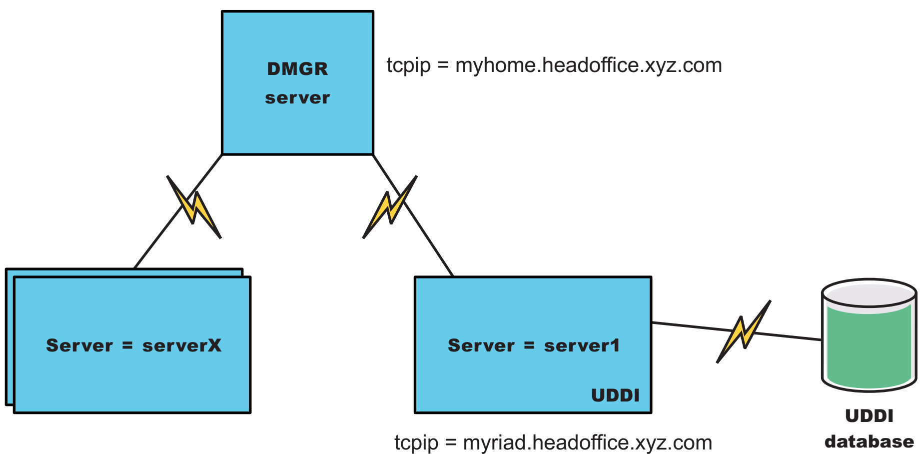
If you intend to run in a Deployment Manager Cell then complete the following task - **Installing the UDDI Registry into a deployment manager cell**

If you intend to run in a single WebSphere Application server, then complete the following task - **Installing the UDDI Registry into a single WebSphere Application Server**

Continue with Configuring the UDDI Registry.

Installing the UDDI Registry into a deployment manager cell

The diagram following shows the configuration used for the Deployment Manager example configurations that follow:



In this configuration, several nodes are federated to the Deployment Manager (dmgr) on myhome.headoffice.xyz.com, from which UDDI is deployed to the host myriad.headoffice.xyz.com.

These instructions assume that the installation has been performed into a clean environment. If you are installing into an existing deployment manager cell skip to step 6.

1. Install the WebSphere Application Server for z/OS product.
2. Install one or more base application servers which will form the cell of servers. One of these should be the application server in which you plan to run an instance of the UDDI Registry. You can run more than one instance of the UDDI Registry within a cell of servers: the UDDI Registry application name is suffixed with the target node and server names to make it unique within the cell (See also "Advanced use of setupuddi.jacl"), but you can only run one UDDI instance within each application server.
3. Ensure that the target application server is stopped.
4. Issue the following command from the `install_root/bin` directory:

   ```
   START dmgr_proc_name,JOBNAME=server_short_name,
         ENV=cell_short_name.node_short_name.server_short_name
   ```

   Note: You must enter this command on a single line. It is split here for display purposes.
5. Run *addNode* (*addNode.sh* on Unix and Linux platforms) on each of the application server(s) to add it as a node into the cell. (How to run addNode is

described elsewhere in the Information Center - see addNode command). For example: addnode myhome- where myhome is the IP name of your deployment manager host.

6. Copy the *uddiejbclient.jar* file and the EJB javadoc directory tree from the UDDIReg/ejb subdirectory of the deployment manager install tree onto any machine(s) where you will be creating EJB clients to access the UDDI Registry.
7. If you have any global configuration properties that are common to any UDDI Registries that you install into this cell, you can edit the *uddi.properties* file in the properties subdirectory of the deployment manager install tree to set them up. (See the section on Configuring global UDDI properties for more details about the global configuration properties).
8. If required, edit the security permissions for the UDDI Registry application. You should only do so if you have a thorough understanding of Java 2 security issues, and the way in which security permissions are used by WebSphere. The permissions for the UDDI Registry application are set within the *was.policy* file, which is part of the *uddi.ear*. To see and change the contents of this file you should:
   a. On the deployment manager, copy the **uddi.ear** file from the installableApps subdirectory of the deployment manager install tree into a temporary directory.
   b. Un-jar the *uddi.ear* file (that is unpack uddi.ear using the 'jar -x' command).

      For example:

      ```
      jar -x uddi.ear
      ```
   c. You will find the *was.policy* file under the META-INF subdirectory that is created.

      This will allow you to see the permissions which have been granted to the UDDI Registry application, and to make any changes that are necessary. Please note that if you make any errors in changing this file, then the UDDI Registry application might either fail to start, or will encounter errors when trying to execute UDDI requests.
   d. Re-jar the *uddi.ear* file using the jar command.

      For example:

      ```
      jar -cf uddi.ear .
      ```

      Note: NOTE the space and the dot after uddi.ear)

      (This uses the jar command in the <DeploymentManager-install-dir>\java\bin subdirectory of the deployment manager, so you might need to fully qualify the path to the jar command.)
   e. Copy the new *uddi.ear* back to the installableApps directory.
9. Note that if the target application server is running, this step will stop and restart it. If you are planning to use Cloudscape for the database in which the UDDI Registry will be held, please read the section "Setting up the UDDI Registry to use Cloudscape within a deployment manager cell" and then return to this point. If however, you plan to use DB2, then please refer to the section "Setting up the UDDI Registry to use DB2 within a deployment manager cell" and then return to this point.
10. Ensure that the UDDI Registry is configured appropriately for your installation, as described in the section on Configuring the UDDI Registry.

11. Start, or stop and restart, the target application server. This should also start the UDDI Registry application. If not, use the administrative console on the deployment manager to do so.

    On z/OS platforms run the db2profile script before issuing the START command. This script is located within the DB2 instance's home directory under SQLLIB and you can invoke it by typing:

    ″. /home/db2inst1/sqllib/db2profile″

    Note: In the above example, notice that the '.' is followed by a single space character.

    Note: On Unix and Linux platforms the DB2 user **must** have a db2profile at $HOME/sqllib/db2profile.
12. Your UDDI application is now ready to use. Go to the User Console section within this Information Center or any of the API sections.

Advanced use of setupuddi.jacl

A number of symbols are defined at the top of the setupuddi.jacl script. These allow you to control the amount of logging that is performed, and to install multiple instances of the UDDI Registry within the same cell.

The symbols that you can edit are as follows:
- **logEnabled** - default setting is 1, which causes the progress of the script to be logged. Setting this symbol to 0 causes information logging to be suppressed, with only error messages being output.
- **overwriteExisting** - default setting is 1 which causes any existing installation of the UDDI Registry application to be overwritten. Setting this symbol to 0 would cause the existing installation to be left as is, but would allow other files used by the UDDI Registry to be updated. You are recommended to only change this setting under the guidance of IBM Service.
- **appName** - default setting is UDDIRegistry, which is the first part of the name used for the UDDI Registry application installed into the target server. To ensure uniqueness of application names within the cell, the full application name that will be used is *<appName>.<nodeName>.<server>*, where *<nodeName>* is the name of the target node and <server> is the name of the target server. You can change the first part of this (the *<appName>*) portion by changing the setting of this symbol before running setupuddi.jacl, although it is generally recommended that you do not change this value.

Continue with Configuring the UDDI Registry.

Setting up the UDDI Registry to use Cloudscape within a deployment manager cell

If you plan to use Cloudscape for the database in which the UDDI Registry data will be held, perform this task to setup and install the UDDI Registry database to use the supplied Cloudscape database.

See ″Choice of database product to be used as the persistence store″ to decide which database product you should use as your persistence store before proceeding further with this task.

This task is part of a parent task: Installing the UDDI Registry into a deployment manager cell. You should complete this task at the appropriate step in the parent task.

This task configures Cloudscape on the host where you want to run the UDDI Registry. Cloudscape is supplied with WebSphere Application Server.

In this task you will invoke a script called setupuddi.jacl, specifying the target node and application server into which the UDDI Registry is to be deployed. If the target application server is running when you invoke setupuddi.jacl, the script stops the server and restarts the server after it has completed its operations.

1. Copy the UDDI20 directory tree from the bin subdirectory of the deployment manager tree into the bin subdirectory of the target application server's installation tree.
2. Create a JDBC driver and datasource to provide access to the UDDI20 Cloudscape database, and install the UDDI Registry application. This is done using the wsadmin tool, using as input the setupuddi.jacl script from the UDDIReg/scripts subdirectory of the Deployment Manager. Note that this script must be run on the deployment manager node.

   You should either run this script from the UDDIReg/scripts subdirectory where it is located, or copy it to some other suitable directory. Note that the wsadmin command is located in the bin subdirectory of the deployment manager node.The syntax for calling this script for Cloudscape is:

   ```
   wsadmin -f setupuddi.jacl
                deploymgrpath
                servername
                nodename
                discoveryURLprefix
                pathtodb
                > setupuddi.log
   ```

   where

   - *deploymgrpath* is the fully qualified pathname of the deployment manager install directory, specified using forward slashes regardless of platform; for example for Windows, this might be C:/Progra~1/WebSphere/DeploymentManager for Windows, or, for Unix platforms it might be /opt/WebSphere/DeploymentManager.
   - *servername* is the name of the target server on which you wish to deploy the UDDI Registry, such as server1. Note the the server name entered is case sensitive.
   - *nodename* is the name of the WebSphere node on which the target server runs. Note the the node name entered is case sensitive.
   - *discoveryURLprefix* is the URL prefix to be used for discovery URLs. Typically this will be of the form http://<ip-address>:9080/uddisoap/ - an example of a discoveryURLprefix value might be http://mynode.mylocation.mycompany.com:9080/uddisoap/
   - *pathtodb* is the path to the UDDI20 database within the bin subdirectory of your WebSphere AppServer installation, specified using forward slashes regardless of platform; for example for Windows, this might be C:/Progra~1/WebSphere/AppServer/bin/UDDI20 and for Unix platforms: /opt/WebSphere/AppServer/bin/UDDI20
   - > *setupuddi.log* is an optional parameter to direct the output to a log file as opposed to the default (which is to the screen)

   For example on Windows (shown here on multiple lines for publication):

```
wsadmin -f setupuddi.jacl "C:/Progra~1/WebSphere/DeploymentManager/" server1 myriad
 "http://myriad.headoffice.xyz.com:9080/uddisoap/" "C:/Progra~1/WebSphere/Appserver/bin/UDDI20"
```

   or, on Unix platforms (shown here on multiple lines for publication):

```
. /wsadmin.sh -f setupuddi.jacl "/opt/WebSphere/DeploymentManager/" server1 myriad
"http://myriad.headoffice.xyz.com:9080/uddisoap/" "/opt/WebSphere/Appserver/bin/UDDI20"
```

installs the UDDI Registry application into the server server1 running on node myriad, and sets it up to access the Cloudscape UDDI20 database located in the bin subdirectory of the application server.

The setupuddi.jacl script:

a. Creates a JDBC driver named *UDDI.JDBC.Driver.<nodeName>.<server>* and a datasource named *UDDI.Datasource.<nodeName>.<server>* (where *<nodeName>* is the name of the target node and *<server>* is the name of the target server, and will replace any existing driver and datasource of that name.
b. Checks whether the UDDI Registry application is already installed and, if so, stop it and uninstall it.
c. Updates the uddi.properties configuration property file to configure the discoveryURLprefix value that you have specified and set the persister property as 'Cloudscape', and place this file into the location *config/cells/<currentcell>/nodes/<nodename>/servers/<servername>/uddi.properties*.
d. Places a number of files that are needed by the UDDI Registry into the WebSphere configuration repository, and updates the ws.ext.dirs list to reference these files.
e. Installs the UDDI Registry.

This script deploys the UDDI Registry into the configuration under the deployment manager, and then do a Synchronization to install it into the specified server.

Note: The setup script, setupuddi.jacl, cannot be used to install the UDDI Registry application into a clustered application server. It is possible to cluster the UDDI Registry application by installing UDDI into an unclustered application server using the setup script, and then cluster that application server.

Return to the next step in the parent task Installing the UDDI Registry into a deployment manager cell.

Setting up the UDDI Registry to use DB2 within a deployment manager cell

To decide which database product you should use as your persistence store, see "Choice of database product to be used as the persistence store".

This task is part of a parent task: Installing the UDDI Registry into a deployment manager cell. You should complete this task at the appropriate step in the parent task.

If you plan to use DB2 for the database in which the UDDI Registry will be held, ensure that the correct prerequisite fix packs have been applied as listed at http://www-3.ibm.com/software/webservers/appserv/doc/latest/prereq.html otherwise the startup of the UDDI DB2 setup wizard will fail.

If you plan to use DB2 for the database in which the UDDI Registry data will be held, use this task to create and load the UDDI Registry database using DB2, and to setup and install the UDDI Registry application to use the DB2 database.

The following steps should be carried out on the system on which the target application server is located (referred to as the 'target system').

In this task you will invoke a script called setupuddi.jacl, specifying the target node and application server into which the UDDI Registry is to be deployed. If the target application server is running when you invoke setupuddi.jacl, the script stops the server and restarts the server after it has completed its operations.

Before starting this task, ensure that you have created an appropriate DB2 userid and password. This same userid and password must be used throughout the following steps where the DB2 userid and password is requested.

1. Create and load the UDDI Registry database.

   Note: 5.0.1 + It is recommended you use the SetupDB2UDDI.jar command from IBM WebSphere Application Server Version 5.0.1 or later. This is essential for non-English users.

   Note: 5.0.1 + If you have a copy of the file SetupDB2UDDI.jar in your application server directory, then the application of the base and Network Deployment PTFs will not update SetupDB2UDDI.jar in your application server directory. You must apply the PTF for Network Deployment to your DeploymentManager file structure to update the SetupDB2UDDI.jar located there (in the /UDDIReg/scripts subdirectory), and then manually copy this jar file to any application server you may wish to run it on.

   If you are planning to use a remote DB2 system on another host machine, copy the SetupDB2UDDI.jar file to the remote system and run it on that system to create and load the UDDI Registry database following the instructions within this step and continue with the next step (which states "If using a remote DB2 system on another host machine..") on the local host.

   5.0.1 + Information on how to do this and where to obtain the PTF can be found here.

   To create the database you use the UDDI DB2 setup wizard, which is supplied as a shell script called SetupDB2UDDI.sh in the UDDIReg/bin subdirectory, by following these steps:

   a. Temporarily set your path by typing:
      - On Windows:
        ```
        set path=%WAS_PATH%;%path%
        ```
      - On Unix or Linux platforms:
        ```
        export PATH=$WAS_PATH:$PATH
        ```
   b. If necessary, check the log files for the wizard. A log file called UDDIloadDB.log is written to the directory from which the wizard is run (but note that on Windows platforms, if you have decided not to overwrite an existing UDDI20 database, then this fact is not logged, and the log file is not be created).
   c. Gather the following information about your DB2 z/OS configuration from your DB2 administrator:
      1) The location of the DB2 system where you want to persist the UDDI directory.

2) The name of the DB2 database that will be used for the UDDI directory. If a database does not already exist, the UDDI DB2 setup wizard will create one for you.
3) The name of the DB2 tablespace that will be used for the UDDI directory. If a tablespace does not already exist, the UDDI DB2 setup wizard will create one for you.
4) The name of the DB2 storage group that will hold the UDDI directory.
5) The name of the volume on which the DB2 database should be created.
6) The name of the alias of the VSAM catalog for DB2 datasets.

d. Run the UDDI DB2 setup wizard

The UDDI DB2 setup wizard supports two modes of operation:

1) Create and load

To initiate this mode of operation, issue the following command from an MVS command line prompt:

```
/install_root/UDDIReg/bin/SetupDB2UDDI.sh DB2_location_name
      DB2_database_name DB2_tablespace_name create
      DB2_storage_group_name volume_name
      VSAM_catalog_name
```

This command should be entered all on a single line. It was split here for formatting purposes.

This mode of operation requires you to specify all of the parameters on the shell script. This enables the DB2 database and tablespace to be created. The newly created database will be primed for use by the UDDI Registry.

Note: If you use the create and load mode of operation, you must run the wizard using a USERID that has the authority to create or update DB2 databases, storage groups, ect. Also, that USERID will be used as the SQLID of the resulting DB2 tables.

2) Load

To initiate this mode of operation, issue the followiing command from an OMVS command line prompt:

```
/install_root/UDDIReg/bin/SetupDB2UDDI.sh DB2_location_name
      DB2_database_name DB2_tablespace_name skip
```

This command should be entered all on a single line. It was split here for formatting purposes.

This mode of operation does not require you to specify the *DB2_storage_group_name volume_name* and *VSAM_catalog_name*parameters on the shell script. This mode uses the existing DB2 database and tablespace and only primes that database for use by the UDDI Registry.

The output from the UDDI DB2 wizard will be placed in a file called setupDB2UDDI.tracelog. Errorsthat occur will be writtento the user's telnet session.

2. If using a remote DB2 system on another host machine, refer to "Use of a remote DB2 database" and then return to this point and continue with the following instructions.
3. Create a JBDC driver and datasource to provide access to the UDDI20 DB2 database, and install the UDDI Registry application. This is done using the

wsadmin tool, using as input the setupuddi.jacl script from the UDDIReg/scripts subdirectory of the deployment manager. This script must be run on the deployment manager node.

Either run this script from the UDDIReg/scripts subdirectory where it is located, or copy it to some other suitable directory. Note that the wsadmin command is located in the bin subdirectory of the deployment manager node.The syntax for this script for DB2 is:

```
wsadmin -f setupuddi.jacl
            deploymgrpath
            servername
            nodename
            discoveryURLprefix
            dbname
            db2ziplocation
            > setupuddi.log
```

where:

- *deploymgrpath* is the fully qualified pathname of the deployment manager install directory, specified using forward slashes regardless of platform; for example for Windows, this might be c:/Progra~1/WebSphere/DeploymentManager and, for Unix platforms it might be /opt/WebSphere/DeploymentManager/
- *servername* is the name of the target application server on which you wish to deploy the UDDI Registry, such as server1. The server name is case sensitive.
- *nodename* is the name of the WebSphere node on which the target application server runs. Typically, this is the same as the machine name. The node name is case sensitive.
- *discoveryURLprefix* is the URL prefix to be used for discovery URLs. Typically this will be of the form *http://<ip-address>:9080/uddisoap/* so an example of a discoveryURLprefix value might be http://mynode.mylocation.mycompany.com:9080/uddisoap/
- *dbname* is the name of the UDDI Registry database under DB2. For this parameter, you should specify the database name you specified when you ran the UDDI DB2 setup wizard.

  Note: If a remote DB2 system is being used the *dbname* stated here must be the alias created when access to the remote database was set up on the local system
- *db2ziplocation* is the path to the db2java zip file on your system, specified using forward slashes regardless of platform; for example for Windows, this might be C:/Progra~1/SQLLIB/java/db2java.zip or, for Unix platforms it might be /home/db2inst1/sqllib/java12/db2java.zip
- > *setupuddi.log* is an optional parameter to direct the output to a log file as opposed to the default (which is to the screen)

For example:

On Windows the command is (shown here on multiple lines for publication):

```
wsadmin -f setupuddi.jacl "C:/Progra~1/WebSphere/DeploymentManager/" server1 myriad
 "http://myriad.headoffice.xyz.com:9080/uddisoap/" UDDI20 db2admin secretpwd
 "C:/Progra~1/SQLLIB/java/db2java.zip" > setupuddi.log
```

On Unix platforms the command is (shown here on multiple lines for publication):

```
wsadmin.sh -f setupuddi.jacl "/opt/WebSphere/DeploymentManager/" server1 myriad
 "http://myriad.headoffice.xyz.com:9080/uddisoap/" UDDI20 db2admin secretpwd
 "/home/db2inst1/sqllib/java12/db2java.zip" > setupuddi.log
```

This installs the UDDI Registry application into the server server1 running on node myriad, and set it up to access the DB2 UDDI20 database using the userid *'db2admin'* and password *'secretpwd'*.

The setupuddi.jacl script:

a. Creatse a JDBC driver named UDDI.JDBC.Driver.<nodeName>.<server> and a datasource named UDDI.Datasource.<nodeName>.<server> (where <nodeName> is the name of the target node and <server> is the name of the target server>, and replaces any existing driver and datasource of that name.
b. Checks whether the UDDI Registry application is already installed and, if so, stop it and uninstall it.
c. Updates the uddi.properties configuration file to configure the discoveryURLprefix value that you have specified, and to set the persister property as *'DB2'*, and places the uddi.properties file into the location config/cells/<currentcell>/nodes/<nodename>/servers/<servername>.
d. Places a number of files that are needed by the UDDI Registry into the WebSphere configuration repository, and update the ws.ext.dirs list to reference these files.
e. Installs the UDDI Registryapplication into the server `server1` running on node `myriad`, and set it up to access the DB2 database..

Note: The setup script, setupuddi.jacl, cannot be used to install the UDDI Registry application into a clustered application server. It is possible to cluster the UDDI Registry application by installing UDDI into an unclustered application server using the setup script, and then cluster that application server.

Return to the next step in the parent task Installing the UDDI Registry into a deployment manager cell.

Installing the UDDI Registry into a single appserver

If you intend to run in a single WebSphere Application server, then complete the following task.

When you select the UDDI Registry option, then the installation will place all files that are needed to run a UDDI Registry onto the deployment manager install tree on the machine on which you are installing IBM WebSphere Application Server for Network Deployment.

To be able to run the UDDI Registry in a single application server instance in your network space you must copy these files over to the application server and then deploy the UDDI Registry. You can do this as follows:

1. Stop the application server on which you plan to run the UDDI Registry; for example, by issuing the following command from the `bin` directory:
```
STOP appserver_proc_name,JOBNAME=server_short_name1,
     ENV=cell_short_name.node_short_name.server_short_name
```
Note: This command must be entered on a single line. It is split here for display purposes.
2. Copy the *uddi.ear* file from the installableApps subdirectory of the deployment manager install tree into the installableApps subdirectory of the target application server's install tree.

3. Copy the *uddi.properties* file from the properties subdirectory of the deployment manager install tree into the properties subdirectory of the target application server's install tree. In a subsequent step, you configure the UDDI Registry using the properties in the *uddi.properties* file.
4. Copy both the *uddiresourcebundles.jar* and the *setupuddimessages.jar* files from the lib subdirectory of the deployment manager install tree into the lib subdirectory of the target application server's install tree.
5. Optionally, if you are going to write or run code that uses the EJB interface to UDDI on another machine, then copy the *uddiejbclient.jar* file and the EJB javadoc directory tree from the UDDIReg/ejb subdirectory of the deployment manager install tree onto a location of your choice on any machines where you will be creating EJB clients to access the UDDI Registry.
6. Configure database support for the UDDI Registry database, in which the UDDI Registry will be held. To do this, complete one of the following tasks then return this point:
   - Setting up the UDDI Registry to use Cloudscape in a single AppServer
   - Setting up the UDDI Registry to use DB2 in a single AppServer

   Note: If you set up the UDDI Registry application with a JDBC driver and datasource that reference Cloudscape, but set the persister property in *uddi.properties* to specify DB2, **or vice versa**, then some unexpected behavior will result, such as a fatal error on deleting an entity. If this happens, you should check that the above details are not in conflict. This only applies to a UDDI Registry installation on a single appserver.
7. Ensure that the UDDI Registry is configured appropriately for your installation, as described in the section on Configuring the UDDI Registry.
8. Stop and then restart the application server. On z/OS platforms you must remember to run the db2profile script before issuing the START *appserver_proc_name* command. This script is located within the DB2 instance's home directory under SQLLIB and can be invoked, for example, by typing:

   ```
   . /home/db2inst1/sqllib/db2profile
   ```

   Note: In the above example, it should be noticed that the '.' is followed by a single space character.

   Note: On Unix and Linux platforms the DB2 user **must** have a db2profile at $HOME/sqllib/db2profile.

Continue with Configuring the UDDI Registry.

Setting up the UDDI Registry to use Cloudscape in a single application server

To decide which database product you should use as your persistence store, see "Choice of database product to be used as the persistence store".

If you plan to use Cloudscape for the database in which the UDDI Registry data is held, use this task to set up and install the UDDI Registry application to use the supplied Cloudscape database.

This task configures Cloudscape on the host where you want to run the UDDI Registry.

This task, to configure Cloudscape for the UDDI Registry database, is part of a parent task Installing and Setting up a UDDI Registry. You should complete this task at the appropriate step in the parent task.

1. Copy the UDDI20 directory tree from the bin subdirectory of the deployment manager tree into the bin subdirectory of the target application server's install tree.
2. Copy the uddicloudscapeuserfunc.jar file from the lib subdirectory of the deployment manager install tree to the lib subdirectory of the target application server's install tree.
3. Ensure that the persister property in the uddi.properties file is set to *persister=Cloudscape*
4. Copy the *appserversetupuddi.jacl* script from the UDDIReg/scripts subdirectory of the deployment manager install tree to the WebSphere Application Server bin subdirectory.
5. Change directory to the WebSphere Application Server bin subdirectory.
6. Start the application server on which the UDDI Registry is to run.

   For example, enter:

   ```
   startServer server1            for Windows
   ```

   ```
   . /startServer.sh server1       for Unix platforms
   ```
7. Create a JDBC driver and datasource to provide access to the UDDI20 Cloudscape database, and install the UDDI Registry application. To do this run the wsadmin tool with the script *appserversetupuddi.jacl* as input, on the target application server, using the following command syntax:

   (You should either run this script from the UDDIReg/scripts subdirectory where it is located, or copy it to some other suitable directory. Note that the wsadmin tool is located in the WebSphere bin subdirectory.)

   ```
   wsadmin -f appserversetupuddi.jacl
          uddi-ear-location
          servername
          nodename
          WebSphere-lib-subdirectory
          cloudscapedbname
          > setupuddi.log
   ```

   Where:

   - *uddi-ear-location* is the fully-qualified path to the uddi.ear file in the installableApps subdirectory, specified **using forward slashes regardless of platform.** For example, on Windows:

     ```
     C:/Progra~1/WebSphere/AppServer/installableApps/uddi.ear
     ```

     and on Unix platforms:

     ```
     /opt/WebSphere/AppServer/installableApps/uddi.ear
     ```
   - *servername* is the name of the application server on which the UDDI registry is to run; for example: `server1`. Note that the server name entered is case sensitive.
   - *nodename* is the name of the WebSphere node on which the application server, *servername*, is running. Typically this is the machine name. Note that the node name entered is case sensitive.
   - *WebSphere-lib-subdirectory* is the fully-qualified path to the WebSphere Application Server lib subdirectory, specified using forward slashes regardless of platform. For example on Windows:

     ```
     C:/Progra~1/WebSphere/AppServer/lib
     ```

     and on Unix platforms:

`/opt/WebSphere/AppServer/lib`

- *cloudscapedbname* is the fully-qualified path to the UDDI20 database within the bin subdirectory of your WebSphere AppServer installation, specified using forward slashes regardless of platform. For example on Windows:

  `C:/Progra~1/WebSphere/AppServer/bin/UDDI20`

  and on Unix platforms

  `/opt/WebSphere/AppServer/bin/UDDI20`
- > *setupuddi.log* is an optional parameter to direct the output to a log file as opposed to the default (which is to the screen)

The appserversetupuddi.jacl script completes the following actions:

a. Creates a JDBC driver named *UDDI.JDBC.Driver.<nodeName>.<server>* and a datasource named *UDDI.Datasource.<nodeName>.<server>* (where *<nodeName>* is the name of the target node and *<server>* is the name of the target server, and will replace any existing driver and datasource of that name.
b. Checks whether the WebSphere UDDI Registry application is already installed and, if so, stop the application and uninstall it.
c. Installs the WebSphere UDDI Registry, then starts it.

Note: The setup script, appserversetupuddi.jacl, cannot be used to install the UDDI Registry application into a clustered application server. It is possible to cluster the UDDI Registry application by installing UDDI into an unclustered application server using the setup script, and then clustering that application server.

Return to the next step in the parent task Installing the UDDI Registry into a single appserver.

Setting up the UDDI Registry to use DB2 in a single application server

To decide which database product you should use as your persistence store, see ″Choice of database product to be used as the persistence store″.

If you plan to use DB2 for the database in which the UDDI Registry is held, ensure that the correct prerequisite fix packs have been applied as listed at http://www-3.ibm.com/software/webservers/appserv/doc/latest/prereq.html otherwise the startup of the UDDI DB2 setup wizard will fail.

If you plan to use DB2 for the database in which the UDDI Registry data will be held, use this task to create and load the UDDI Registry database using DB2, and to setup and install the UDDI Registry application to use the database.

This task uses the UDDI DB2 setup wizard to configure DB2 on the system where you want to run the UDDI Registry. Before starting this task, ensure that DB2 is installed and running on that system.

Copy the UDDIReg directory tree from the deployment manager to the target application server where DB2 will run.

The following steps should be carried out on the system on which the target application server is located (referred to below as the ’target system’).

1. Create and load the UDDI Registry database.

Note: 5.0.1 + It is recommended you use the setupDB2UDDI.jacl command from IBM WebSphere Application Server Version 5.0.1 or later. This is essential for non-English users.

Note: 5.0.1 + If you have a copy of the file SetupDB2UDDI.jar in your application server directory, then the application of the base and Network Deployment PTFs will not update SetupDB2UDDI.jar in your application server directory. You must apply the PTF for Network Deployment to your DeploymentManager file structure to update the SetupDB2UDDI.jar located there (in the /UDDIReg/scripts subdirectory), and then manually copy this jar file to any application server you may wish to run it on.

If you are planning to use a remote DB2 system on another host machine, copy the SetupDB2UDDI.jar file to the remote system and run it on that system to create and load the UDDI Registry database following the instructions within this step and continue with the next step (which states ″If using a remote DB2 system on another host machine ....″) on the local host.

To create the database you use the UDDI DB2 setup wizard:

a. Change directory to the directory containing the file SetupDB2UDDI.sh (that is, either the UDDIReg/bin subdirectory, or a directory on the target system into which you have copied it).
b. To run the wizard, you need to first ensure that you have access from your command line to the JVM supplied with WebSphere. This is done as follows:
   - At a command line, enter one of the following commands:
     - If you are using bash:

       ```
       . <AppServer-install-dir>/bin/setupCmdLine.sh
       ```
     - If you are using csh:

       ```
       source /<AppServer-install-dir>/bin/setupCmdLine.sh
       ```
c. Temporarily set your path by typing:
   - On Windows:

     ```
     set path=%WAS_PATH%;%path%
     ```
   - On Unix or Linux platforms:

     ```
     export PATH=$WAS_PATH:$PATH
     ```
d. In the same command window, start the UDDI DB2 setup wizard by issuing the following command:

   ```
   SetupDB2UDDI.sh
   ```
e. Follow the prompts to work through the wizard panels or command prompts.
f. If necessary, check the log files for the wizard. A log file called UDDIloadDB.log is written into the directory from which the wizard is run (but note that, on Windows platforms, if you have decided not to overwrite an existing UDDI20 database, this fact is not logged, and the log file is not created).

2. If using a remote DB2 system on another host machine, refer to ″Use of a remote DB2 database″ and then return to this point and continue with the next step.
3. Ensure that the persister property in the uddi.properties file is set to `persister=DB2`.

4. On Unix, run the db2profile script to set up the environment for the DB2 instance that the UDDI Registry is using:

```
. /home/db2inst1/sqllib/db2profile
```

5. Start the application server on which the UDDI Registry is to run. For example:

   On Windows:

   On Unix platforms:

```
START appserver_proc_name,JOBNAME=server1,
      ENV=cell_short_name.node_short_name.server1
```

   Note: This command must be entered on a single line. It is split here for display purposes.

6. Copy the *appserversetupuddi.jacl* script from the UDDIReg/scripts subdirectory of the deployment manager install tree to the WebSphere Application Server bin subdirectory.
7. Change directory to the WebSphere Application Server bin subdirectory.
8. Create a JDBC driver and datasource to provide access to the database, and install the UDDI Registry application. To do this run the wsadmin tool with the script *appserversetupuddi.jacl* as input, on the target application server, using the following command syntax. (Either run this script from the UDDIReg/scripts subdirectory where it is located, or copy it to some other suitable directory. Note that the wsadmin tool is located in the WebSphere bin subdirectory.)

```
wsadmin -f appserversetupuddi.jacl
             uddi-ear-location
             servername
             nodename
             WebSphere-lib-subdirectory
             dbname
             db2userid
             db2pwd
             db2ziplocation
             > setupuddi.log
```

   where

   - *uddi-ear-location* is the fully-qualified path to the uddi.ear file in the installableApps subdirectory, specified **using forward slashes regardless of platform**.

     For example, on Windows:

     and on Unix platforms

     `/opt/WebSphere/AppServer/installableApps/uddi.ear`
   - *servername* is the name of the application server on which the UDDI registry is to run; for example: `server1`. Note that the name of the server is case sensitive.
   - *nodename* is the name of the WebSphere node on which the application server, *servername*, is running. Typically, this will be the same as the machine name. Note that the name of the node is case sensitive. Typically this is the machine name.
   - *WebSphere-lib-subdirectory* is the fully-qualified path to the WebSphere Application Server lib subdirectory, specified **using forward slashes regardless of platform**. For example:
     – On Windows: `C:/Progra~1/WebSphere/AppServer/lib`
     – On Unix: `/opt/WebSphere/AppServer/lib`
     – On Unix or z/OS: `/opt/WebSphere/AppServer/lib`
   - *dbname* is the name of the UDDI Registry database under DB2. You should specify UDDI20 for this parameter

Note: If a remote DB2 system is being used the *dbname* must be the alias created when access to the remote database was set up on the local system.

- *db2userid* and *db2pwd* are a valid DB2 userid and password with administrative privileges, as specified in an earlier step.
- *db2ziplocation* is the path under which you have installed DB2, specified **using forward slashes regardless of platform.**. For example, for Windows, this might be C:/Progra~1/SQLLIB/java/db2java.zip or, for Unix platforms it might be /opt/SQLLIB/java//db2java.zip.
- > *setupuddi.log* is an optional parameter to direct the output to a log file as opposed to the default (which is to the screen)

The appserversetupuddi.jacl completes the following actions:

a. Creates a JDBC driver named *UDDI.JDBC.Driver.<nodeName>.<server>* and a datasource named *UDDI.Datasource.<nodeName>.<server>* (where *<nodeName>* is the name of the target node and *<server>* is the name of the target server>, and will replace any existing driver and datasource of that name.
b. Checks whether the WebSphere UDDI Registry application is already installed and, if so, stop the application and uninstall it.
c. Installs the WebSphere UDDI Registry, then starts it.

Note: The setup script, appserversetupuddi.jacl, cannot be used to install the UDDI Registry application into a clustered application server. It is possible to cluster the UDDI Registry application by installing UDDI into an unclustered application server using the setup script, and then clustering that application server.

Return to the next step in the parent task Installing and Setting up a UDDI Registry.

Reinstalling the UDDI Registry application

If you wish to reinstall the UDDI Registry, follow the appropriate section below.

Reinstalling into a deployment manager cell

If you wish to reinstall the UDDI Registry into the target application server, for example because you wish to alter certain aspects of its configuration using AAT, rerun the setupuddi.jacl script (described in the appropriate link as follows):

- ″Setting up the UDDI Registry to use Cloudscape within a deployment cell″
- ″Setting up the UDDI Registry to use DB2 within a deployment cell″

Note: If you decide to change from using Cloudscape as your persistence store to DB2, or vice versa, first remove UDDI from the application server using removeuddi.jacl. You can then run setupuddi.jacl to reinstall UDDI with the new type of persistence store.

If you make such a change, then any data that you had previously stored is no longer be accessible.

Reinstalling into a single appserver

Remove the UDDI Registry application in the same manner as any other Enterprise Application and then install as described in the appropriate link:

- Setting up the UDDI Registry to use Cloudscape in a single AppServer
- Setting up the UDDI Registry to use DB2 in a single AppServer

Note: If you decide to change from using Cloudscape as your persistence store to DB2, or vice versa, first remove UDDI from the application server using appserverremoveuddi.jacl. You can then run appserversetupuddi.jacl to reinstall UDDI with the new type of persistence store.

If you make such a change, then any data that you had previously stored is no longer accessible.

Removing the UDDI Registry application from a deployment manager cell

To completely remove the UDDI Registry application from the target application server in the deployment manager cell, run the *wsadmin* (*wsadmin.sh* on Unix Platforms) script *removeuddi.jacl*, which is located in the UDDIReg/scripts directory of the deployment manager install tree.

If the target server specified on invoking removeuddi.jacl is running at the same time, the script stops the server and restarts the server when it has completed its operations.
At a command promp enter:

```
wsadmin -f removeuddi.jacl
            servername
            nodename
            > removeuddi.log
```

Where *servername* and *nodename* are the server and node where you have deployed the UDDI Registry application. By default output will go to the screen, but, optionally, you can specify '> removeuddi.log' to direct output to a log file. For example,

```
wsadmin -f removeuddi.jacl server1 myriad
```

will remove the UDDI Registry application and related files from server server1 running in node myriad, and will send any messages to the screen.

Removing the UDDI Registry application from a single application server

To completely remove the UDDI Registry application from a stand-alone application server run the wsadmin script appserverremoveuddi.jacl, which was installed into the UDDIReg/scripts directory when you installed the UDDI Registry as part of a Network Deployment install.
At a command prompt enter:

```
wsadmin -f appserverremoveuddi.jacl
              servername
              nodename
              > removeuddi.log
```

where
- *servername* and *nodename* are the name of the stand-alone application node in which it runs (these are the names that you specify when you run appserversetupuddi.jacl to install the UDDI Registry application).

- by default output will go to the screen, but, optionally, you can specify '> *removeuddi.log*' to direct the output to a log file.

For example:

```
wsadmin -f appserverremoveuddi.jacl server1 monolith
```

will remove the UDDI Registry application and related files from server server1 running in node monolith, and will send any messages to the screen.

Configuring the UDDI Registry

The UDDI Registry is supplied as a J2EE application file, uddi.ear. This is installed into the WebSphere Application Server during installation. If you want to change any of its configuration properties using Application Assembly Tool (AAT) see "Configuring SOAP properties with the AAT".

You can configure the following aspects of the UDDI Registry:
- Configuring global UDDI properties
- Modifying the database userid and password
- Configuring security properties
- Configuring the UDDI User Console (GUI) for multiple language encoding support
- Customizing the UDDI User Console (GUI)
- Configuring SOAP interface properties
- 5.0.1 5.0.2 + Configuring SOAP properties with the Application Assembly Tool
- Configuring SOAP properties in the deployment descriptor

Configuring global UDDI properties

To modify any of the global UDDI properties, edit the file called `uddi.properties`. More than one version of this file exists and the version you need to edit depends on:
- whether you are in the installation phase or are updating the properties as a post installation step
- whether you are configured for a deployment manager or base application server environment

The location of the file you should edit will be one of the following:

Deployment Manager Configurations

1. If you are in the process of installing the UDDI Registry application for the first time into a deployment manager cell and wish to make some generic changes before deploying it in the cell, the *uddi.properties* file will be located in the <DeploymentManager-install-dir>/properties directory. If you are reinstalling the UDDI Registry application into a deployment manager cell, then you should edit the file in the location described in step 2.

   Note: In a deployment manager configuration some properties (such as persister and getServletURLprefix) are dynamically set up in the *uddi.properties* file, during subsequent installation processing.
2. If the UDDI Registry is already configured into an application server within a Deployment Manager cell (that is you are undertaking post installation configuration changes), the *uddi.properties* file you should edit is located in the configuration repository, under the deployment manager filing system; that is

in <*DeploymentManager-install-dir*>*config/cells/*<*cellname*>*/nodes/*<*nodename*>*/servers/*<*servername*>, where <*cellname*> is the name of the deployment manager cell, <*nodename*> is the name of the node in which the application server is installed, and <*servername*> is the name of the application server in which you have installed the UDDI Registry.

Application Server Configurations

1. If you are in the process of installing the UDDI Registry application into an application server only environment you will be advised during the installation process when to make changes to the *uddi.properties* file.

   Note: In contrast with the deployment manager configuration, UDDI properties are not dynamically set during installation processing.
2. If the UDDI Registry is already configured into a single application server that is **not** part of a deployment manager cell (i.e. you are undertaking post installation configuration changes), then the *uddi.properties* file will be located in the properties subdirectory of the WebSphere Application Server in which you have installed the UDDI Registry application, that is <ApplicationServer-install-dir>/properties directory.

The properties that can be changed within *uddi.properties* are as follows:

- The dbMaxResultCount, which is the limit on the number of rows of information that should be returned on Find requests, and will apply if the request does not specify a maxRows limit itself (or if it specifies a limit that exceeds this value). The initial value for this in *uddi.properties* is 100.
- The persister, which indicates what database is to be used as the persistence store for the UDDI Registry database. If you have installed the UDDI Registry into an application server within a deployment manager cell, then the persister property will have been set to the correct value for you. If you change this value, you must also ensure that you have a UDDI Registry database created using the chosen database product (for more details about the UDDI Registry database, refer to the section on "Installing the UDDI Registry"). You should also be aware that any data published to the UDDI Registry with one setting of the persister property will **not** be accessible when running the UDDI Registry application with a different setting for the persister property. The valid values for the persister property are:
  – persister=DB2

    indicating that DB2 is to be used as the persistence store
  – persister=Cloudscape

    indicating that Cloudscape is to be used as the persistence store

  The initial value for this in *uddi.properties* is Cloudscape.

  Note: This property is dynamically set by the setupuddi.jacl script when installing into a deployment manager cell so in this case you should not need to modify it.
- The default language to be used on a publish request as the xml:lang attribute when one is not specified. The initial value for this in *uddi.properties* is en-US. This property must contain one of the valid xml:lang values.
- The UDDI site operator name. This is a string that is stored in every registry object, to indicate the operator of the UDDI Registry. The initial value for this in *uddi.properties* is `www.mycompany.com/uddi`. This property does not have any particular functional use, so its value can be set to any string that you feel is suitable.

- The maximum number of search keys that can be used on find API requests. The initial value for this in `uddi.properties` is 5.
- The getServletURLprefix and getServletname name, used to build up the discovery URL. The initial values for these are `http://localhost:9080/uddisoap/` and `get`. If you have installed the UDDI Registry into an application server within a deployment manager cell, then the getServletURLPrefix property will have been set for you using the value you specified as a parameter to the setup script. You are recommended to set suitable values for these properties before you first use the UDDI Registry.

  Note: This property is dynamically set by the *setupuddi.jacl* script when installing into a deployment manager cell so in this case you should not need to modify it.

Applying these changes to your system

For your changes to take effect, you must do one of the following:
- If you are in the process of installing the UDDI Registry application for the first time, return to the original topic and complete the installation steps. Any changes you have made are picked up during this subsequent processing.
- If you have made post installation changes in a base application server only environment, you should stop and restart the UDDI Registry application using the WebSphere administrative console.
- If you have made post installation changes in a deployment manager environment you should:
  1. Run a Full Resynchronization for the node where the UDDI Registry runs. This can be done from the WebSphere Network Deployment Administrative Console under section Systems Administration ==> Nodes. Select your node, and then click 'Full Resynchronization'.

     It is important that you do a 'Full Resynchronization' and not just a 'Synchronize'.
  2. Stop and restart the UDDI Registry application using the WebSphere administrative console

Modifying the database userid and password

If you use DB2 as the persistence store for the UDDI Registry, and you need to change the database userid and/or password, alter the user and password values in the custom properties of the 'UDDI Datasource', which can be edited from the WebSphere administrative console. The UDDI.Datasource is under datasources within the UDDI.JDBC.Driver, which is itself found under JDBC Providers under Resources. Do not alter the databaseName.

Configuring security roles

Each interface to the UDDI Registry (either through SOAP, EJB or the GUI) is supplied with two roles:

Publish role
: mapped to AllAuthenticatedUsers. By default, this is configured to use SSL (that is HTTPS), but this only applies when WebSphere security is enabled.

Inquiry role
: mapped to Everyone. By default, this is configured to use HTTP (that is not SSL).

The security roles and use of SSL can be altered by users through the Administration Console.

Authentication uses the standard WebSphere facilities and there is no separate registration function for the Registry. If WebSphere security is enabled, the you will need to supply your WebSphere userid and password for Publish functions (unless you have changed the supplied Publish role).

You will need to set up WebSphere security configuration to be used by UDDI. For information on achieving this, refer to Configuring Secure Sockets Layer: WebSphere Application Server within this Information Center. It is expected that, for development use, security will be disabled and security will be enabled for production environments.

The SOAP interface also supports the UDDI API for get_authToken and discard_authToken API but use of this is optional.

- If security is disabled and get_authToken is not called, the default user, UNAUTHENTICATED, is used.
- If security is disabled and get_authToken is called, the specified userid is used (but the password is not checked).
- If WebSphere security is enabled, it takes priority over UDDI authentication, but if the Publish role is mapped to Everyone, get_authToken must be used and the userid and password will be checked by WebSphere.

The Security Roles provided with the UDDI Registry are as follows:

- GUI_Publish_User
- GUI_Inquiry_User
- SOAP_Publish_User
- SOAP_Inquiry_user
- EJB_Inquiry_Role
- EJB_Publish_Role

Configuring the UDDI User Console (GUI) for multiple language encoding support

If you want to use multiple language encoding support in the User Console (GUI), you must configure the application server into which the UDDI Registry application is installed with UTF-8 encoding enabled. To do this, refer to ″Configuring application servers for UTF-8 encoding″ elsewhere in the WebSphere Information Center.

Customizing the UDDI User Console (GUI)

The look and feel of the UDDI console is determined by the styles defined in the *uddi_gui.css* file which is located in the /gui.war/theme directory of the installed UDDI Registry application directory. The UDDI Registry application directory will be one of the following, depending on where you have installed the UDDI Registry:

- If you have installed the UDDI Registry into an application server within a deployment manager cell, the directory is *install_root*/installedApps/*current_cell*/UDDIRegistry.*node*.*server*.ear/gui.war
- If you have installed the UDDI Registry into a single application server which is not part of a deployment manager cell, the directory is UDDIRgistry.ear under

the installedApps directory of the WebSphere Application Server in which you have installed the UDDI registry application as shown in the example below.

The contents of this file can be edited to change the colors, fonts and font sizes according to the user's preference.

The content and layout of the UDDI User Console is provided by Java Server Pages (JSP), which can be customized by a programmer who is familiar with JSPs. The JSP pages are found in the uddi.ear enterprise application, which is under the installedApps subdirectory of the WebSphere AppServer installation. To locate the JSPs, expand the UDDI_Registry.ear, open the gui.war, and they are located under WEB-INF in the pages subdirectory. So, on a Windows system that has WebSphere installed in the default location, the JSP files will be found in

```
<AppServer-install-dir>\installedApps\<nodename>\UDDI_Registry.ear\gui.war\WEB-INF\pages
```

These JSP pages also contain some application logic (as opposed to presentation logic) that should not be changed.

Configuring SOAP interface properties

You can configure the following SOAP interface properties:
- *defaultPoolSize* - the number of SOAP parsers with which to initialize the parser pool for the SOAP interface. You can set this independently for the Publish (uddipublish) and Inquiry (uddi) APIs. For example, if you expect more inquiries than publish requests via the SOAP interface, you can set a larger pool size for the Inquiry API. The default initial size for both APIs is 10.
- The *context root* used for the Publish and Inquiry APIs, which forms a part of the URL by which they are accessed. By default this is /uddisoap.
- Whether the API is to be secure (via HTTPS) or insecure (via HTTP). The default is to use HTTPS.

To configure the following SOAP interface properties, use either of the following methods:
- Configuring SOAP properties with the Application Assembly Tool (the recommended option, especially for a production environment)
- Configuring SOAP properties in the deployment descriptor for the SOAP module in the UDDI application directly. This option is faster and may be the preferred method in a test environment.

Configuring SOAP properties with the Application Assembly ToolWebSphere Assembly Toolkit or the Application Assembly Tool

To configure SOAP properties by using the Application Assembly ToolWebSphere Assembly Toolkit or the Application Assembly Tool, complete the following steps:
- Select *Update* and click on the Application icon.
- Select the *uddi.ear* file (this is placed, by the UDDI installation, into the UDDI install directory (e.g. *C:\WebSphere\installableApps\uddi.ear*).
- Expand the *uddi.ear* icon on the left-hand pane in the AAT.
- Expand the *Web Modules* tree.
- Expand the *uddi Soap* tree
- To change the *defaultPoolSize*, expand Web Components and then *uddipublish* (for the publish API) or *uddi* (for the inquiry API).

- Click on *Initialization Parameters* which will show the *defaultPoolSize* parameter in the upper right-hand pane. This can be edited in the lower right-hand pane.

- To change the *context root*, click on *UDDI Soap* which displays general information about the SOAP module in the lower right-hand pane in AAT. The *context root* can be edited in this pane.
- To change the publish API to use HTTP (instead of HTTPS), click on *Security Constraints* and change the *Transport Guarantee* from Confidential to none.
- Having made any changes above, you must now save them. To do this, click on *File -> Save (or Save As)* to save your changes.
- Redeploy the *uddi.ear* to WebSphere, by first removing it and reinstalling it via the Administrator's Console.

Configuring SOAP properties in an application that is already deployed

To configure SOAP properties after the UDDI application has been installed:

1. Edit the deployment descriptor for the SOAP module (*web.xml)*. This file is located in the *WEB-INF* subdirectory of the *uddi.ear* application in the installed applications within the WebSphere install directory (for example, *<WebSphere-install-dir>\installableApps\uddi.ear\soap.war\WEB-INF*).
2. Stop and restart the application server for the changes to take effect.

Administering the UDDI Registry

Perform the following tasks to administer the UDDI Registry:

- Running the UDDI Registry
- Backing up and restoring the UDDI Registry database

Running the UDDI Registry

Starting the WebSphere Application Server in which the UDDI Registry is deployed

After a reboot, or at any time required, the server can be started by running startServer (on Windows) or startServer.sh (on Unix and Linux platforms).

On Unix and Linux platforms run the db2profile script before issuing the startServer.sh server1 command. This script is located within the DB2 instance home directory under sqllib and is invoked by typing:

```
. /home/db2inst1/sqllib/db2profile
```

Note: In the above example, notice that the '.' is followed by a single space character.

Note: On Unix and Linux platforms the DB2 user **must** have a db2profile at $HOME/sqllib/db2profile

By default, the UDDI Registry is started automatically when the application server is started. In order to stop and restart it, use the administrative console.

You can also use the administrative console to change this default behavior.

Backing up and restoring the UDDI Registry database

If you want to protect the data in your UDDI Registry database, you can back up and restore the database using the facilities of the database product. For DB2, you can do this by using the export and import utilities of the DB2 Control Center. For Cloudscape you can simply use operating system tools to copy the database directory. Refer to the database product information for more details.

DB2 allows for dynamic backup, but, if you are using static backup, stop any UDDIReg applications beforehand.

The UDDI Registry database is called UDDI20, and the tables that should be backed up are:
- ADDRESS
- ADDRLINE
- BSERVICE
- BTEMPLATE
- BUSINESS
- CATEGORY
- CATEGORYBAG
- CONTACT
- DESCR
- DISCOVERYURL
- EMAIL
- EXTCATEGORY
- IDENTIFIERBAG
- INSTANCEDETAIL
- NAMEELEMENT
- OVERVIEWDOC
- PHONE
- PUBLISHERASSERTION
- SERVICEPROJECTION
- TMODEL
- VALIDATIONCACHE
- VALIDATIONSERVICES

UDDI user console

This topic describes the layout of the UDDI user console (also referred to as the Graphical User Interface (GUI)), which you can use to interact with the IBM WebSphere UDDI Registry.

For information about how to display the UDDI user console, see Displaying the user console.

If you will be using the UDDI console, it is recommended that you configure the application server into which you have installed the UDDI Registry for UTF-8 encoding support: see ″Configuring the UDDI User Console for multiple language encoding support″.
- The user console provides a graphical user interface to the majority of the UDDI Version 2 API. It is not intended to support the full API set: there is some focus on inquiry operations, as the main purpose of the UDDI user console is to allow users to issue inquiry requests and to familiarize themselves with general UDDI

concepts. This section documents those areas for which support through the user console is not provided, together with other known restrictions to the user console.
  – General
    - Help is provided in the form of explanatory text on the screens.
    - Maximum rows cannot be specified on finds. The single maximum rows value for the registry can be set through the *dbMaxResultCount* global configuration property. For more information on setting this property see Configuring global UDDI properties
  – Find business
    - The business identifier feature is not supported.
  – Find service type
    - The business identifier feature is not supported.
  – Add business
    - You must supply the business contact as a name and role (no other information is supported).
  – Add service type
    - You can enter the overview URL, but only with one description in English.
  – Add service
    - There is no support for entering a Hosting Redirector, nor for adding an overviewURL.
- **Note:** The UDDI Version 2 specification states that when a tModel is deleted, it should not be physically deleted. This allows the tModel to be reinstated. One effect of this is that, if you delete a tModel using the UDDI user console, the tModel is still visible through the Show Owned Entities display.

The UDDI user console is split into three distinct areas. At the top of the screen are buttons that activate various functions in the areas below this bar. These buttons are:

Home Returns you to the IBM WebSphere UDDI Registry welcome page

Find Activates the Find tab on the frame below to the left

Publish
Similarly activates the Publish tab on the frame below to the left

Below the WebSphere UDDI Registry banner the screen is split into two parts. On the left are the two tabs mentioned above, the Find and Publish tabs.

Find tab

The Find tab is in two parts. At the top, a **Quick Find** service is provided. There are three radio buttons to enable a choice of ’service’, ’business’ and ’technical model’ finds. Below these radio buttons is a text entry box for entering the name to search for and, beneath this, a ’Find’ link to start the search. Comments are provided to show the user the wildcard character. The results of clicking on the ’Find’ link are shown in the detail frame to the right.

Beneath the Quick Find is a section for **Advanced Find** functions which enables the user to choose which entity they want to perform an advanced search on. There are three links: Find services, Find businesses and Find technical models. Clicking one of these links displays the corresponding advanced search form in the frame to the right, which the user may use to enter search criteria. To initiate a Find, the user must first enter a search path (the % wildcard may be used) and then click the blue Add link to enter the search. Then click on the ’Find Services’ (or ’Find Businesses/Find technical models) link below to initiate the Find operation.The **Locator** section has a link (marked in blue with the words ″**Show category tree**″) which displays the tree from which the user can select categories

(or taxonomies). This is shown in the left-hand frame. In the advanced search form there are two links to start the search (mid-way down and at the bottom).

The results of clicking either of the two links to start the search are displayed in the same detail frame.

Publish tab

The *Publish* link on the top banner activates the Publish tab in the navigation frame to the left. The Publish tab is split into three distinct sections.

1. **Quick Publish Function**

   The top part is a **Quick Publish** section to allow the user to publish a business or technical model by name only. There are two radio buttons to enable a choice of 'business' or 'technical model'. Below these radio buttons is a text entry box for entering the name to assign to the selected entity and, beneath this, a blue 'Publish now' link to publish the entity. The results of clicking on the **Publish now** link are shown in the detail frame to the right.
2. **Advanced Publish Functions**

   To publish an entity with more detail, such as with multiple names, descriptions and categories, use the **Advanced Publish** section below this. The comments below each link ('Add a business' and 'Add a technical model') describe individual functions. Clicking one of these links displays the corresponding advanced publish form in the detail frame where the user may enter details about the entity they want to publish. As in the Advanced Find functions described above, there are two links to publish a business or technical model (one towards the top of the form and the other at the bottom). Similarly the **Locator** section allows taxonomies to be shown in the left frame from which the user can select categories.

   Following entry of the relevant details on the **Advanced Publish** section, the user must click on the **Publish Business** bar in order for the business to be published to the UDDI Registry.
3. **Registered Information**

   Below the Advanced Publish section is a **Registered Information** section which has a link to **Show Owned Entities** in order to show the businesses, services and technical models registered to the individual user, and pending business relationships. Clicking the **Show Owned Entities** link displays the **Show Owned Entities** page in the detail frame at the right. The **Show Owned Entities** page is organized in three sections: **Registered Businesses**, **Pending Business Relationships** and **Registered Technical Models**. Each section shows the number of registered items.

   Edit and Delete Businesses

   Users can **Edit** or **Delete** businesses owned by them by clicking the appropriate links in the **Actions** column.

   After an **Edit** or **Delete** function has been completed, the user **must** click on the **Update Business** bar in order to publish the changes to the UDDI Registry.

   After Deleting a Business the user must confirm the deletion by clicking on the 'Delete this Business' link.

   Adding a Service to a Business

   Services are added to a business by clicking the **Add a Service** link in the **Services** column of the **Registered Businesses** section.

   After the **Add a Service** function is complete, users **must** click on the **Publish Service** bar in order to publish the service to the UDDI Registry.

   Referencing a Service from a Business

Services can also be 'referenced' by a business as if the business was the owner of the service. This 'service projection' is performed by clicking the **Reference a service** link in the **Services** column. Services associated with a business, whether they are owned or referenced, can be displayed by clicking the **Show services** link. This acts as a toggle between displaying services available for editing or deleting, and hiding them.

Adding a relationship to another Business

A business can be associated with another business in the UDDI Registry and this function is performed by clicking the **Add a relationship** link in the **Actions** column of the Registered Businesses section. Clicking the **Show related businesses** link in the **Actions** column displays a list of any completed business relationships.

The **Pending Business Relationships** section shows all incomplete publisher assertions, where only one party has asserted a relationship and is waiting for the other party to make the same assertion. This section reminds the user of any relationships that involve their businesses. Once both parties have asserted the same relationship between two businesses, the relationship moves from the **Pending Business Relationships** section and appears in the list of relationships displayed after clicking the **Show related businesses** link in the **Registered Businesses** section.

Technical Models

Technical Models owned by the user are shown in the bottom **Registered Technical Models** section. As for businesses, users can Edit or Delete technical models owned by them by clicking the appropriate links in the **Actions** column.

Note: Users should take note that deletion of Technical Models (tModels) does **not** cause them to be physically deleted, but hidden. This is in accordance with the UDDI Registry V2.0 specifications. After deletion Technical Models are shown under the "**Shown Owned Entities**" link on the publish page but not via the Find links on the Find page. ALL other entities are deleted from the UDDI Registry in the normal way.

Example of publishing a Business, Service and tModel with the User Console

For the example, here, we will assume a business called Mondeo Cars that sells used cars

1. <u>**Add the Business**</u>

   Click on the Publish tab in the left hand navigation frame. Then click on 'Add a business' in the Advanced Publish in the left pane. This takes you to a 'Publish Business' pane on the right. Start by adding your Business Name in the text field labelled (Mondeo Cars in this example) and select a language and then click on the blue Add link to the right. This adds the business name (but the business is not yet published - more about which is explained later). Below the Business Name is an area called Descriptions - it allows free text to be added to describe the business - if you enter anything here you must click on the blue Add link to the right to insert the description.

   The next section/area is the Locator area which can be used to describe the business according to what categories it falls into. This example uses a Used car dealership. Within the NAICS taxonomy (which you may view by clicking on 'Show category tree' and then expanding NAICS) this is a Retail Trade [44] entry which, on expansion, has Motor Vehicle and Parts Dealers [441] and, again on expansion, has automobile Dealers [4411] and Used Car Dealers [44112]. This fits the Business perfectly, so clicking on Used Car Dealers will

enter the Key Name and Key Value into the business. For Checked Categorizations (such as NAICS) the Key name is not checked but the Key Value is checked. It should be noticed that for unchecked categorizations (such as ’other’ or unspsc’) the Key value is not checked either. If the locator field has been added, then the blue Add link must be clicked. The final area is Contacts, which can have names and role information added if required. Again, the blue Add link must be clicked after adding the relevant information.

Once all the fields are filled in to the required level, the final action is to click on the Publish Business at the bottom of the form or at the top. This causes the business to be published to the UDDI Registry and a page is displayed showing the business details.

2. **Add a Service**

   From the Publish tab, there is a ’Show owned entities’ link. This shows the businesses owned by the current user in the Registry and the language to be used for a particular user. For Mondeo Cars, the user will see a ’Add service’ button. Clicking this button shows the Publish Service form. The top part of the form is the Service Name field. After adding this name, the user must click on the blue Add link to enter the name. As in the Adding the Business form, each subsequent part must end with the blue Add link being clicked to add that part of the information to the service. The sections are (from top to bottom, Description (a free text area), Access Points (to add link points to the Service), Locator (to add references to taxonomies to the service), and Technical Models (to associate existing tModels to the Service). After completion of those areas required, clicking on the ’Publish Service’ button will Publish the service to the UDDI Registry with the current form contents.

3. **Adding a new technical model**

   Clicking on the ’Add a technical model’ link in the left frame opens up the Publish Technical Model form on the right. A tModel can only have one name - hence the lack of a blue Add link next to the Technical Model Name field. Beneath this field are other fields - Description (a free text area to describe the technical model), Locator (to describe the technical model with taxonomies, and an Overview URL (which gives a URL pointing to an overview document, a description of the document and a Language field). For each of these fields there is a blue Add link which must be clicked to add the relevant data. At the bottom of the form is a ’Publish Technical Model’ link which will create the technical model in the UDDI Registry.

   There is a Publish link at the top of the frame in each case also - after the Name section.

- Displaying the user console

Displaying the user console

Access without authentication enabled

This topic describes how to display the UDDI Registry user console (also referred to as the GUI). By default two URLs are supplied, one for inquiry (non-SSL) and one for publish (via SSL). This section describes the default behavior.

- For inquiry you can access the UDDI User Console by using the following URL in your Web browser:

  http://<hostname>:9080/uddigui
- For publish you can access the UDDI user console by using the following URL in your Web browser:

  https://<hostname>:9443/uddigui

Note: With WebSphere security disabled, all the publish operations are performed using a userid of UNAUTHENTICATED. Also, if you select a publish action on the GUI you will automatically be redirected to the SSL port.

Access with authentication enabled.

If you have WebSphere security enabled, you can access the UDDI User Console through the two URLs as above, however, for publish request, you will be prompted for a WebSphere uid and password.

The user console displays the default frameset containing the header frame, navigation frame showing find options, and details frame. When you click the link to show the publish options in the navigation frame, you are challenged for a userid and password.

If WebSphere security is enabled and you try to access a publish action via an unsecured link, e.g. clicking the publish link on the navigation frame where the user console was opened with:

http://<hostname>:9080/uddigui

you are redirected to a secure logon screen. Inquire functions work as expected.

The uDDI Registry supports a number of security roles, including two for the user console. See Configuring Security Roles within this Information Center for more details on this topic.

Custom Taxonomy Support in the UDDI Registry

The IBM WebSphere UDDI Registry is supplied with six published taxonomies (or categorization schemes) in the taxonomy data. Taxonomies can be either checked or unchecked, and this is indicated via a keyedReference in the categoryBag of the tModel that represents a taxonomy (a ″categorization tModel″). These keyedReferences have the tModel key for uddi-org:types and are added to the categoryBag to further describe the behavior of the categorization tModel, as follows:

checked
: Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that has a validation service to check that category values are present in a specified value set.

unchecked
: Marking a tModel with this classification asserts that it represents a categorization, identifier, or namespace tModel that does not have a validation service.

Of these six published taxonomies, four are checked.

In the IBM WebSphere UDDI Registry and also in the IBM UDDI Business Registry (UBR), the validation of categories in checked taxonomies is performed against locally managed taxonomy data. The published taxonomies are:

Taxonomy name	Checked	Description	tModel key

ntis-gov:naics:1997	Yes	Business Taxonomy: NAICS (1997 Release)	uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2
uddi-org:iso-ch:3166-1999	Yes	ISO 3166-1:1997 and 3166-2:1998. Codes for names of countries and their subdivisions. Part 1: Country codes. Part 2: Country subdivision codes. Update newsletters include ISO 3166-1 V-1 (1998-02-05), V-2 (1999-10-01), ISO 3166-2 I-1 (1998)	uuid:4E49A8D6-D5A2-4FC2-93A0-0411D8D19E88
unspsc-org:unspsc	Yes	Product Taxonomy: UNSPSC	uuid:CD153257-086A-4237-B336-6BDCBDCC6634
unspsc-org:unspsc:3-1	No	Product Taxonomy: UNSPSC (Version 3.1)	uuid:DB77450D-9FA8-45D4-A7BC-04411D14E384
uddi-org:types	Yes	UDDI Type Taxonomy	uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4
uddi-org:general_keywords	No	Special taxonomy consisting of namespace identifiers and the keywords associated with the namespaces	uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4

Taxonomy data is provided in the IBM WebSphere UDDI Registry for all the above taxonomies, apart from the general keywords taxonomy (which is unchecked). The UDDI User Console (GUI) provided with the IBM WebSphere UDDI Registry uses a shortened label for taxonomies when displayed in the taxonomy tree view, or in a pull-down list of available taxonomies as follows:

Taxonomy Name (published)	Taxonomy name (as displayed in the UDDI user console)
ntis-gov:naics:1997	naics
uddi-org:iso-ch:3166-1999	geo
unspsc-org:unspsc	unspsc7
unspsc-org:unspsc:3-1	unspsc
uddi-org:types	udditype
uddi-org:general_keywords	other

This release of IBM WebSphere UDDI Registry (included with IBM WebSphere Application Server, Version 5.0.2) introduces the ability to add user-defined taxonomies, with available allowed values presented in the existing GUI taxonomy tree display. IBM WebSphere Studio Application Developer, Version 5.1 has a Web Services Explorer user interface that also allows addition and display of custom checked taxonomies. The publisher of a custom taxonomy's categorization tModel may specify a 'display name' for use in GUI implementations.

Procedure for adding a Custom taxonomy

To add a custom taxonomy to the IBM WebSphere UDDI Registry requires you to perform two tasks: load the custom taxonomy data and publish a categorization tModel. Only when both are complete will the checked taxonomy be of practical use. Taxonomy data must be provided for validating checked taxonomies.

Taxonomy data *may* also be used by GUIs for unchecked taxonomies, but it is not a requirement and is usually only used for presentation of deprecated taxonomies, such as unspsc-org:unspsc.

If the taxonomy is checked, then any publish requests that have a categoryBag containing keyedReferences with the new categorization tModel will be validated. If there is taxonomy data corresponding to the categorization tModel in the registry database then only valid values will be accepted. If there is no taxonomy data in the database then **all** values will be rejected, and the publish request will fail. If the categorization tModel is unchecked, all values will be allowed, regardless of whether there is corresponding taxonomy data present in the UDDI Registry database.

Suggested approach

The suggested way of introducing a new taxonomy is to:
1. Load custom taxonomy data into the UDDI Registry database using the UDDITaxonomyTools.jar utility (described below)
2. Publish the categorization tModel with a keyedReference of type 'general keywords' with keyname of 'customTaxonomy:key' and a keyValue matching the taxonomy name in the taxonomy data file (described below also)

Note: the SOAP and EJB interfaces will be able to make use of categorization tModels as soon as they are published. However, the UDDI Registry GUI will currently require a restart of the UDDI application because it currently gathers its list of categorizations for use in the taxonomy tree display when the application starts.

Loading Custom Taxonomy Data

Custom Taxonomy Data File Format

Taxonomy data is identified by a common taxonomy name, a unique code value, an optional description and a parent code that specifies its relationship with other code values. Taxonomy data must adhere to this format:

Column name	Maximum length	Description of use
name	8	Uniquely identifies the taxonomy within the registry

code	32	Unique value within the taxonomy used for validation
description	128	Typically used by GUIs and optionally in the keyedReference as the keyName value
parentcode	32	Indicates which existing **code** is the logical parent of this one, and is used in tree displays

Typically columns are delimited in the taxonomy data file by '#' characters as in this example:

```
food#00#Food#00
food#10#Fruit#00
food#101#Apples#10
food#102#Oranges#10
food#103#Pears#10
food#1031#Anjou#103
food#1032#Conference#103
food#1033#Bosc#103
food#104#Pomegranates#10
food#20#Vegetables#00
food#201#Carrots#20
food#202#Potatoes#20
food#203#Peas#20
food#204#Sprouts#20
```

In the example, 'Food' is the description for the root node with child nodes of 'Fruit' and 'Vegetables' (both of these have parentcode values the same as the code value for 'Food').

The taxonomy data in the example file could then be rendered in a tree like this:

```
Food
  Fruit
    Apples
    Oranges
    Pears
      Anjou
      Conference
      Bosc
    Pomegranates
  Vegetables
    Carrots
    Potatoes
    Peas
    Sprouts
```

The file must be saved in UTF-8 format.

The following taxonomy names are reserved within the IBM WebSphere UDDI Registry and should not be used for custom taxonomy files: **naics, geo, unspsc, unspsc7, other, udditype**. Any attempts to publish a categorization tModel using these values for a customTaxonomy:key are rejected. If these names are used in custom data files and the data is imported it is indistinguishable from taxonomy data with the same name.

UDDITaxonomyTools.jar

A utility is provided to load taxonomy data into the IBM WebSphere UDDI Registry, rename existing taxonomy data and remove existing taxonomy data, for both IBM DB2 and Cloudscape databases. The usage for each database and platform is identical:

```
Usage: java -jar UDDITaxonomyTools.jar {function} [options]

function:
  -load <path>          Load taxonomy data from specified file
  -rename <old> <new>   Rename existing taxonomy
  -unload <name>        Unload existing taxonomy

options:
  -properties <path>    Specify location of configuration file
```

Note: Ensure that the command window from which the UDDITaxonomyTools.jar is run is using a suitable codepage and font for displaying the characters contained in the taxonomy name.

Use of an incorrect codepage/font may result in unclear messages on a successful load, and create difficulty using the -unload and -rename options.

UDDITaxonomyTools.jar is located, by default, in the <DeploymentManager-install-dir>/UDDIReg/scripts directory.

The following section explains in more detail how to use the utility's commands and parameters. The configuration file, if specified by the optional properties parameter, determines the database driver, authentication information and delimiters. The contents are as follows (typical data for DB2 installation shown):

Property and example data (for DB2)	Comments
classpath= ″c:/program files/sqllib/java12/db2java.zip; c:/tools/UDDITaxonomyTools.jar″	Classpath including database driver *and* the UDDITaxonomyTools.jar*
database.driver.className=COM.ibm.db2.jdbc.app.DB2Driver	Fully qualified classname of the database driver class
database.url=jdbc:db2:UDDI20	JDBC URL of the database
database.userName=db2admin	Database userid (DB2 only)
database.password=db2admin	Database password (DB2 only)
column.delimiter=#	Column delimiter used in taxonomy data files
string.delimiter=\″	Field delimiter (must be different to the column.delimiter value)

* the classpath needs to be enclosed in quotes if the path includes space characters. Also, the UDDITaxonomyTools.jar filepath itself must be appended to the classpath (if the working directory is the same as the location of the UDDITaxonomyTools.jar then just the name is sufficient)

Filepath names should include the use of the forward-slash character (/) for all platforms.

For Cloudscape database users, the values of the following properties would be likely to be:
- classpath=c:/websphere/appserver/lib/db2j.jar; UDDITaxonomyTools.jar
- database.driver.className=com.ibm.db2j.jdbc.DB2jDriver
- database.url=jdbc:db2j:c:/ websphere/appserver/bin/uddi20

The string.delimiter is typically used where a description value contains the same character as the column delimiter character. For example, if the column.delimiter was set to ',' (comma), and there was a taxonomy description value of 'Fruits, citrus', you could include this in the taxonomy data file by setting the

string.delimiter to "(double quote) and enclosing the description in quotes: 'Fruits, citrus'. Note that the quote character is escaped with a backslash to indicate the literal character is to be used.

If a properties parameter is not specified, the utility looks for and uses configuration data set in a file called customTaxonomy.properties. This file is located, by default, in the <DeploymentManager-install-dir>/UDDIReg/scripts directory.

Note: to make updates to taxonomy data in a Cloudscape database, the IBM WebSphere Application Server must be stopped to release the connection to the database.

Note: There is currently a limitation with UDDITaxonomyTools.jar when used with a DB2 UDDI database and multi-byte characters such as Chinese, Japanese and Korean. The maximum number of multi-byte characters is the maximum value specified earlier for name, code, description and parentcode divided by 3. For example, name can only contain values up to 8 characters in length so the maximum number of Korean characters is 2. If the taxonomy file is found to have values that exceed the limits, a message is displayed by the tool indicating the line number and column where the problem occurs. This limitation does not affect use with a Cloudscape UDDI database.

Publishing a Checked Categorization tModel

This section describes how to publish a checked categorization tModel with the 'customTaxonomy' keyedReferences to specify which custom taxonomy data to use and a display name.

Note: to specify an unchecked categorization substitute the 'checked' keyValue with 'unchecked' or, more simply, omit the keyedReference.

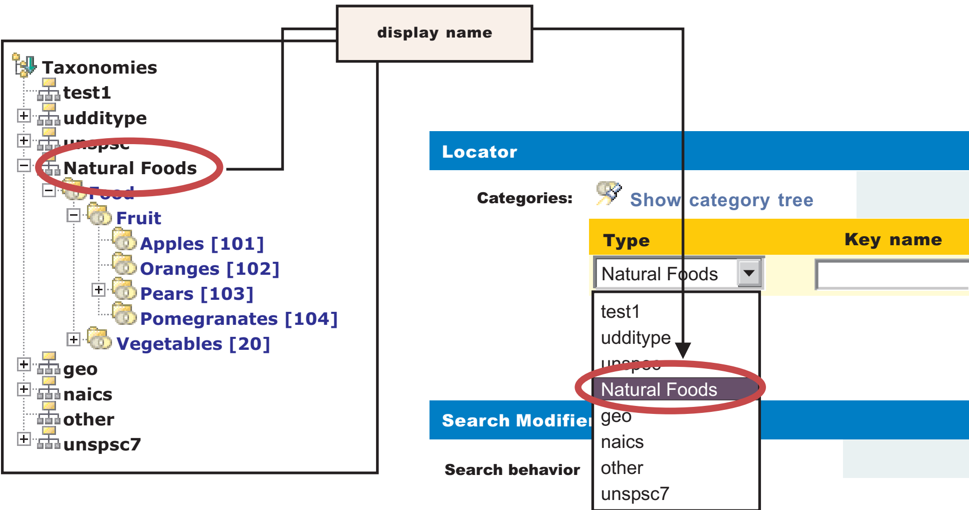
Publish a tModel to the IBM WebSphere UDDI Registry with a categoryBag containing keyedReferences as follows:

Note	tModelKey	KeyName	KeyValue
1	(uddi-org:types)	<optional>	categorization
2	(uddi-org:types)	<optional>	checked
3	(general keywords)	urn:x-ibm:uddi:customTaxonomy:key	*<custom taxonomy name>*
4	(general keywords)	urn:x-ibm:uddi:customTaxonomy:displayName	*<custom taxonomy displayName>*

1. Indicates this tModel is a categorization tModel (required)
2. Indicates use of the tModel will be checked against a list of valid data (required). (Omitting this keyedReference, or explicitly specifying a value of 'unchecked' will indicate this categorization is unchecked).
3. Indicates special use of the general keywords taxonomy, with a proprietary urn as the keyName value, defines the value used by the UDDI Registry to look up taxonomy data in its database. The value must be 1-8 (inclusive) characters long and corresponds directly with the name value in the custom taxonomy data file. Therefore, it must be unique within the registry.
4. Indicates special use of the general keywords taxonomy, with a proprietary urn as the keyName value, defines a name for the custom taxonomy that is

intended for use in GUI implementations where the full tModel name might be too long*. The value can be 1-255 characters (inclusive) long. If this keyedReference is not supplied, the name of the tModel should be used by the GUI implementation.

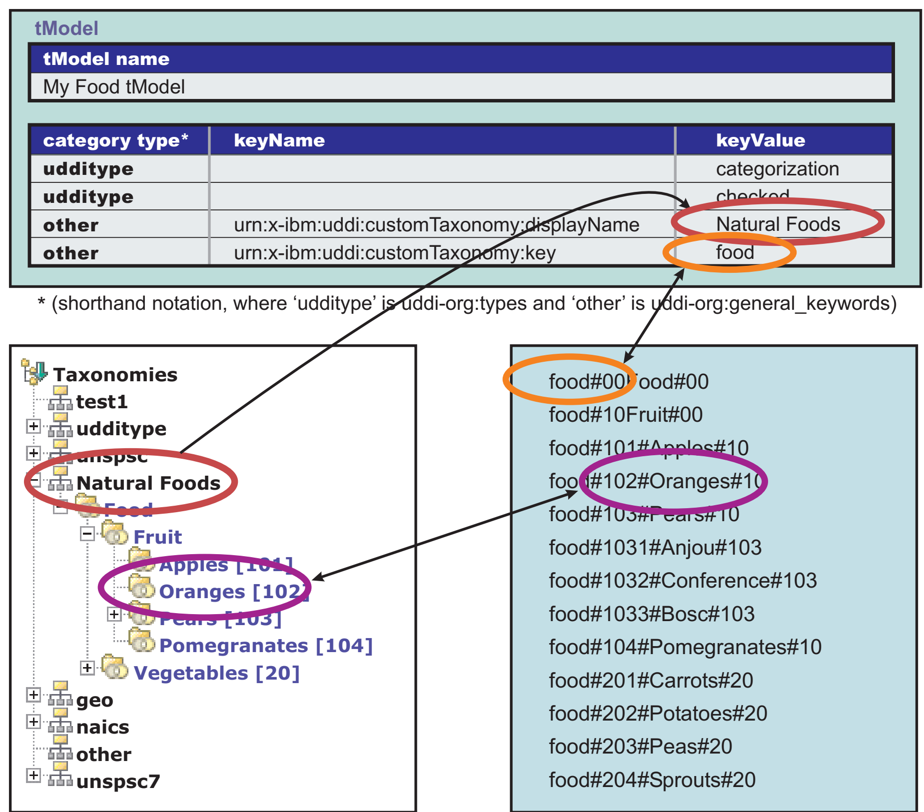
* The displayName is intended to provide a way to label a taxonomy such that, when the UDDI GUI displays it in a taxonomy tree or in a pull-down list of available taxonomies, the meaning is clear to the user without being restricted to 8 characters and without needing to be the same as the published tModelName, which could be as long as 255 characters. An example is shown:



Uniqueness of the urn:x-ibm:uddi:customTaxonomy:key value is validated at the time a categorization tModel is published. If it is not unique, a UDDIInvalidValueException is returned. If using a GUI to publish the tModel, an appropriate message is displayed indicating the likely cause of the problem.

The urn:x-ibm:uddi:customTaxonomy:displayName should be unique if only to avoid confusion when displayed in GUIs but this is not validated.

The relationship between the various keyedReferences, the custom taxonomy data files and use in GUIs for a categorization tModel is shown below:



As a further example, to display the label 'Delicious Victuals' in GUI displays, the categorization tModel would have a keyedReference like this:

type	keyName	keyValue
other	urn:x-ibm:uddi:customTaxonomy:displayName	Delicious Victuals

And to link a categorization tModel to a custom taxonomy datafile with a taxonomy name of 'goodfood' the tModel's categoryBag must have a keyedReference like this:

type	keyName	keyValue
other	urn:x-ibm:uddi:customTaxonomy:key	goodfood

To publish a new categorization tModel using SOAP, the message would be:

```
<save_tModel generic="2.0" xmlns="urn:uddi-org:api_v2">
  <authInfo></authInfo>>
  <tModel tModelKey="">
    <name>Natural Foods tModel</name>
    <categoryBag>
      <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
                      keyValue="categorization"/>
      <keyedReference tModelKey="uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4"
                      keyValue="checked"/>
      <keyedReference tModelKey="uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4"
                      keyName="urn:x-ibm:uddi:customTaxonomy:key"
                      keyValue="food"/>
```

```
      <keyedReference tModelKey="uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4"
                      keyName="urn:x-ibm:uddi:customTaxonomy:displayName"
                      keyValue="Natural Foods"/>
    </categoryBag>
  </tModel>
</save_tModel>
```

Note: ’uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4’ is the tModel key for uddi-org:types and ’uuid:A035A07C-F362-44DD-8F95-E2B134BF43B4’ is the tModel key for uddi-org:general_keywords.

Validation and Error Handling

For a DB2-based IBM WebSphere UDDI Registry, custom taxonomy data can be loaded, removed and renamed using the provided utility without restarting the application (if you are using Cloudscape the application server must be stopped to make database updates). Removing data for which there is a corresponding checked categorization tModel will cause any use of that categorization’s data to be reported as invalid.

Note: If an attempt is made to add data with a name that matches any of the ’internal’ taxonomies, such as naics, geo, and so on, the request is rejected. If an attempt is made to rename or remove one of the internal taxonomies, a warning message is returned. Likewise if the user tries to rename a taxonomy to one of the reserved taxonomies, that is rejected.

The UDDI Registry user console performs validation while a save tModel request is being built, that is, before the publish occurs. For example, if a categorization tModel with a customTaxonomy:key keyValue of ’food’ already exists (in a published categorization tModel), and the user tries to add a keyedReference with the same value to the current list of keyedReferences, the following message is displayed:

```
Advice: The 'urn:x-ibm:uddi:customTaxonomy:key' value of 'food' is already
        in use by another categorization tModel. Enter a unique value
```

Similarly, only one of each of the customTaxonomy:key and customTaxonomy:displayName keyedReferences are allowed. For example, if the user tries to add two customTaxonomy:displayName keyedReferences the following message is displayed:

```
Advice: Only one 'urn:x-ibm:uddi:customTaxonomy:displayName' key name is
        allowed for the 'Other' taxonomy
```

If the customTaxonomy:key keyedReference is valid and unique at the time it is added to the save_tModel request, the keyedReference is further validated when the user makes the publish request, to ensure that another session has not successfully published a categorization tModel with the same customTaxonomy:key. In this case, the user is returned to the Publish Technical Model page.

If a keyedReference containing a keyName value that starts with ’urn:x-ibm:uddi:customTaxonomy:’ is followed by anything other than ’key’ or ’displayName’, the following message is displayed:

```
Advice: Only key name values of 'urn:x-ibm:uddi:customTaxonomy:displayName'
        and 'urn:x-ibm:uddi:customTaxonomy:key' are supported.
```

For SOAP, UDDI4J, and EJB initiated requests where the save_tModel message may have multiple tModels, if any one of the tModels is a categorization tModel

and it fails validation, the request fails with a UDDIInvalidValueException (plus additional information explaining the likely cause), and none of the tModels is published. For example, if a publish request includes a customTaxonomy:key keyedReference with a keyValue that matches the customTaxonomy:key keyValue of an existing categorization tModel, the following UDDIInvalidValueException is thrown, with the message:

```
E_invalidValue (20200) A value that was passed in a keyValue attribute did
 not pass validation. This applies to checked categorizations, identifiers
 and other validated code lists. The error text will clearly indicate the
 key and value combination that failed validation.
 Invalid 'customTaxonomy:dbKey' keyValue [naics] in keyedReference. KeyValue
 already in use by tModelKey[UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2]
```

The customTaxonomy:key and customTaxonomy:displayName keyValue values are validated. For example, a publish categorization tModel request with a keyedReference including a customTaxonmy:key of 'toolongdbkey' was attempted, the following UDDIInvalidValueException is thrown, with the message:

```
E_invalidValue (20200) A value that was passed in a keyValue attribute did
 not pass validation.  This applies to checked categorizations, identifiers
 and other validated code lists. The error text will clearly indicate the
 key and value combination that failed validation. Invalid
 'customTaxonomy:key' keyValue [toolongdbkey] in keyedReference. tModelKey[]
```

If a categorization tModel is edited in the user console, or republished via SOAP, UDDI4J or EJB, such that it is no longer a categorization tModel (ie the categorization keyedReference is removed), then that tModel is removed from the internal store of categorization tModels, and its customTaxonomy:key value, if present, is available for use by new categorization tModels.

SOAP application programming interface for the UDDI Registry

Access to the SOAP API will by default be available at:

http://localhost:9080/uddisoap/inquiryapi

or

https://localhost:9443/uddisoap/publishapi

Where 'localhost' is the address by which your WebSphere server is known. If security is enabled on your WebSphere server, the publishapi will also be protected by basic-authentication. By default, when security is enabled, the publishapi is restricted to HTTPS, this is to ensure the confidentiality and security of your data whilst in transit to UDDI. If you do not wish to use SSL, when security is enabled, modify the jar file using AAT to remove the CONFIDENTIAL restriction placed upon the publish URLs. For more information about this topic, see the section on **Configuring SOAP properties with the Application Assembly Tool**. If you normally access your WebSphere server via a Web server, ensure the plugin configuration for the WebSphere plugin on the Web server has been updated since installing UDDI. This allows access to the UDDI SOAP API through the URLs :

http://localhost/uddisoap/inquiryapi

or

https://localhost/uddisoap/publishapi

Where ’localhost’ is the address by which your Web server is accessed. Note that if you plan on accessing UDDI via a Web server in this manner, that the samples will require modification to inform them of the SSL certificates used by your Web server, so that the samples can make SSL connections to the Web server. It is beyond the scope of this document to cover the many variants available on Web server/WebSphere/java SSL configurations

- Using the SOAP API
- Handling Errors as a User of the SOAP API

Programming the SOAP API

To use the SOAP API construct a properly formed UDDI message within the body of a SOAP request, and send it using HTTP POST to the URL of the API that the request relates to. The response is returned within the body of the HTTP reply. Although the samples are written in Java, you can use other programming languages to create your SOAP client, providing you still send requests compliant to the SOAP specification. Valid UDDI requests should conform to the UDDI schema, and be as detailed within the UDDI standard documentation:

http://www.uddi.org/

For more information on using the SOAP API, refer to ″The UDDI Registry application programming interface″.

SOAP API error handling tips in the UDDI Registry

When using the SOAP API there are three main categories that can cause an error to be returned:

- An incorrect request being sent to the SOAP API., for example:
  - incorrectly formed XML
  - badly formed UDDI requests
  - non-schema compliant requests
- Incorrect business logic within a SOAP API request, for example attempting to delete a business that does not exist.
- Problems occurring while processing a valid request., for example server connection to database failure.

In each of these cases, an error is returned to the client that made the request, which attempts to explain further what the problem was.

UDDI Registry Application Programming Interface

The IBM WebSphere UDDI Registry fully supports the application programming interface (API) specification, which can be viewed by visiting `http://www.uddi.org/pubs/ProgrammersAPI_v2.pdf`. Any changes from this specification are documented within the IBM WebSphere UDDI Registry information.

- The Inquiry API
- The Publish API

Inquiry API for the UDDI Registry

The Inquiry API provides four forms of query that follow broadly used conventions that match the needs of software traditionally used within registries.

- The browse pattern
- The drill-down pattern
- The invocation pattern
- Inquiry API functions

Browse pattern for the UDDI Registry

Software that allows people to explore and examine data - especially hierarchical data - requires browse capabilities. The browse pattern characteristically involves starting with some broad information, performing a search, finding general result sets and then selecting more specific information for drill-down.

The UDDI API specifications accommodate the browse pattern by way of the *find_xx* API calls. These calls form the search capabilities provided by the API and are matched with summary return messages that return overview information about the registered information that is associated with the inquiry message type and the search criteria specified in the inquiry.

A typical browse sequence might involve finding whether a particular business you know about has any information registered. This sequence would start with a call to *find_business*, perhaps passing the first few characters of a business name that you already know. This returns a *businessList* result. This result is overview information (keys, names and descriptions) derived from the registered businessEntity information, matching on the name fragment that you provided. If you spot the business you are looking for within this list, you can drill down into the corresponding businessService information, looking for particular technical models (for example purchasing, shipping, and so on) using the *find_service* API call. Similarly, if you know the technical *fingerprint* (tModel signature) of a particular software interface and want to see if the business you have chosen provides a Web service that supports that interface, you can use the find_binding inquiry message.

Drilldown pattern for the UDDI Registry

When you have a key for one of the four main data types managed by a UDDI registry, you can use that key to access the full registered details for a specific data instance. The UDDI data types are businessEntity, businessService, bindingTemplate and tModel. You can access the full registered information for any of these structures by passing a relevant key type to one of the *get_xx* API calls.

Continuing the example from the Browse pattern for the UDDI Registry, one of the data items returned by all of the *find_x* return sets is key information. In the case of the business we were interested in, the businessKey value returned within the contents of a businessList structure can be passed as an argument to *get_businessDetail*. The successful return to this message is a *businessDetail* message containing the full registered information for the entity whose key value was passed. This will be a full *businessEntity* structure.

Invocation pattern for the UDDI Registry

To prepare an application to take advantage of a remote Web service that is registered within the UDDI registry by other businesses or entities, you must prepare that application to use the information found in the registry for the specific service being invoked.

The *bindingTemplate* data obtained from the UDDI registry represents the specific details about an instance of a given interface type, including the location at which

a program starts interacting with the service. The calling application or program should cache this information and use it to contact the service at the registered address whenever the calling application needs to communicate with the service instance. In previously popular remote procedure technologies tools have automated the tasks associated with caching (or hard coding) location information. Problems arise however when a remote service is moved without any knowledge on the part of the callers. Moves occur for a variety of reasons, including server upgrades, disaster recovery, and service acquisition and business name changes.

When a call fails using cached information previously obtained from a UDDI Registry, the proper behavior is to query the UDDI Registry for fresh bindingTemplate information. If the data returned is different from the cached information, the service invocation should automatically retry the invocation using the fresh information. If the result of this retry is successful, the new information should replace the cached information.

By using this pattern with Web services, a business using a UDDI Registry can automate the recovery of a large number of partners without undue communication and coordination costs. For example, if a business has activated a disaster recovery site, most of the calls from partners fail when they try to invoke services at the failed site. By updating the UDDI information with the new address for the service, partners who use the invocation pattern automatically locate the new service information and recover without further administrative action.

Inquiry API functions in the UDDI Registry

These messages represent inquiries that can be made of the UDDI Registry. These messages all behave synchronously.

The queries available are:

find_binding
: Locates specific bindings within a registered businessService. Returns a bindingDetail message that contains zero or more bindingTemplate structures matching the criteria specified in the argument list.

find_business
: Locates information about one or more businesses. Returns a businessList message that matches the conditions specified in the arguments.

find_relatedBusinesses
: Locates information about businessEntity registrations that are related to a specific business entity whose key is passed in the inquiry. The Related Businesses feature is used to manage registration of business units and subsequently relate them based on organizational hierarchies or business partner relationships. Returns a relatedBusinessList message containing results that match the conditions specified in the arguments.

find_service
: Locates specific services within a registered businessEntity. Returns a serviceList message that matches the conditions specified in the arguments.

find_tModel
: Locates a list of tModels that match a set of specified criteria. The response will be a list of abbreviated information about registered tModel data that matches the criteria specified. The result will be returned in a tModelList message.

get_bindingDetail
: Requests the run-time bindingTemplate information for the purpose of invoking a registered business API. Returns a bindingDetail message.

get_businessDetail
: Returns complete businessEntity information for one or more specified businessEntity registrations matching on the businessKey values specified. Returns a businessDetail message.

get_businessDetailExt
: Returns extended businessEntity information for one or more specified businessEntity registrations. This message returns exactly the same information as the get_businessDetail message, but may contain additional attributes if the source is an external registry with the API specification.

get_serviceDetail
: Requests full information about a known businessService structure. Returns a serviceDetail message.

get_tModelDetail
: Gets full details for a given set of registered tModel data. Returns a tModelDetail message.

Publish API for the UDDI Registry

The messages in this section represent commands that are used to publish, delete and update information contained in a UDDI registry. The messages defined in this section all behave synchronously.

The Publishing API calls defined that UDDI operators support are:

add_publisherAssertions
: Causes one or more publisherAssertions to be added to an individual publisher's assertion collection.

delete_binding
: Causes one or more instances of bindingTemplate data to be deleted from the UDDI registry.

delete_business
: Removes one or more business registrations and all direct contents from a UDDI registry.

delete_publisherAssertions
: Causes one or more publisherAssertion elements to be removed from a publisher's assertion collection.

delete_service
: Removes one or more businessService elements from the UDDI registry and from its containing businessEntity parent.

delete_tModel
: Logically deletes one or more tModel structures. Logical deletion hides the deleted tModels from find_tModel result sets but does not physically delete them, so they are returned on a get_registeredInfo request.

discard_authToken
: Informs an operator site that the authentication token is to be discarded, effectively ending the session. Subsequent calls that use the same authToken will be rejected. This message is optional for operator sites that do not manage session state or that do not support the get_authToken message.

get_assertionStatusReport
: Provides administrative support for determining the status of current and outstanding publisher assertions that involve any of the business registrations managed by the individual publisher account. Using this message, a publisher can see the status of assertions that they have made, as well as see assertions that others have made that involve businessEntity structures controlled by the calling publisher account.

get_authToken
: Obtains an authentication token.Authentication tokens are opaque values that are required for all other publisher API calls. This message is not required for operator sites that have an external mechanism defined for users to get an authentication token. This API is provided for implementations that do not have some other method of obtaining an authentication token or certificate, or that choose to use userid and password based authentication.

get_publisherAssertions
: Obtains the full set of publisher assertions that are associated with an individual publisher account. Publisher assertions are used to control publicly visible business relationships.

get_registeredInfo
: Gets an abbreviated list of all businessEntity and tModel data that are controlled by the individual associated with the credentials passed.

save_binding
: Saves or updates a complete bindingTemplate element. this message can be used to add or update one or more bindingTemplate elements as well as the container/contained relationship that each bindingTemplate has with one or more existing businessService elements.

save_business
: Saves or updates information about a complete businessEntity element. This API has the broadest scope of all the save_xx API calls in the publisher API, and can be used to make sweeping changes to the published information for one or more businessEntity elements controlled by an individual.

save_service
: Adds or updates one or more businessService elements exposed by a specified businessEntity.

save_tModel
: Adds or updates one or more registered tModel elements.

set_publisherAssertions
: Manages all of the tracked relationship assertions associated with an individual publisher account.

For full details of the syntax of the above queries, refer to the API specification at `http://www.uddi.org/pubs/ProgrammersAPI_v2.pdf`.

UDDI EJB Interface for the UDDI Registry

This section describes how to use the EJB application programming interface (API) of the IBM WebSphere UDDI Registry component to publish, find and delete UDDI entries.

The necessary client classes are contained in the *uddiejbclient.jar* file in the ejb subdirectory of the UDDIReg directory under the WebSphere application server directory tree.

The Javadoc for the EJB API is contained in the javadoc directory tree under the ejb subdirectory of the UDDIReg directory under the WebSphere appserver directory tree.

The EJB API is contained in two stateless session beans, one for the Inquiry API (com.ibm.uddi.ejb.InquiryBean) and one for the Publish API (com.ibm.uddi.ejb.PublishBean), whose public methods form an EJB interface for

the UDDI Registry. All the public methods on the InquiryBean correspond to UDDI Inquiry API functions, and all the public methods on the PublishBean correspond to UDDI Publish API functions. (Not all UDDI API functions are implemented, for example get_authToken, discard_authToken, get_businessDetailExt, and so on) For Version 1 of the UDDI registry, the EJB component supports only UDDI v2.0.

The two EJBs use container-managed transactions. The transaction attribute for the methods of the InquiryBean is NotSupported, and for the methods of the PublishBean it is Required. You must not change the transaction attributes as this could result in undesirable behavior.

Within each interface there are groups of overloaded methods that correspond to the operations in the UDDI 2.0 specification. There is a separate method for each major variation in function. For example, the single UDDI 2.0 operation find_business is represented by 10 variations of findBusiness methods, with different variations for finding by name, finding by categoryBag and so on.

The arguments for the EJB interface methods are java objects in the package com.ibm.uddi.datatypes. Roughly speaking, there is a one-one correspondence between classes in this package and elements of the UDDI V2.0 XML schema. Exceptions to this are, for example, where UDDI XML elements can be represented by a single String. (See Package com.ibm.uddi.datatypes below for more information.)

Enabling an EJB Client

This section is written on the assumption that WebSphere Application Server V5.0, a supported database and the IBM WebSphere UDDI Registry have already been installed.

Classpaths

Add the following jar files and folders to your CLASSPATH:

For Windows

<WebSphere-install-dir>\lib\j2ee.jar
<WebSphere-install-dir>\lib\naming.jar
<WebSphere-install-dir>\lib\namingclient.jar
<WebSphere-install-dir>\lib\ecutils.jar
<WebSphere-install-dir>\lib\sas.jar
<WebSphere-install-dir>\properties

For Unix Platforms, and also including z/OS

<WebSphere-install-dir>/lib/j2ee.jar
<WebSphere-install-dir>/lib/naming.jar
<WebSphere-install-dir>/lib/namingclient.jar
<WebSphere-install-dir>/lib/ecutils.jar
<WebSphere-install-dir>/lib/sas.jar
<WebSphere-install-dir>/properties

In addition to these jars, there is also the jar file that contains all of the UDDI specific API for the EJB interface, which can be found at:

For Windows

```
<DeploymentManager-install-dir>\UDDIReg\ejb\uddiejbclient.jar
```

where <DeploymentManager-install-dir> is the install location for WebSphere Application Server for Network Deployment, which by default is C:\Progra~1\WebSphere\DeploymentManager.

For Unix Platforms, and also including z/OS

```
<DeploymentManager-install-dir>/UDDIReg/ejb/uddiejbclient.jar
```

where <DeploymentManager-install-dir> is the install location for WebSphere Application Server for Network Deployment, which by default is */opt/WebSphere/DeploymentManager* for Linux/Solaris systems or */usr/WebSphere/DeploymentManager* for AIX systems.

The Path

Ensure that your PATH statement starts with <WebSphere-install-dir>\java\bin

Creating an EJB Client

If you want to read about creating EJB Clients in more detail, then please read the ″Sun Microsystems Enterprise JavaBeans™ Specification Version 2.0″

Finding the EJB Reference

An EJB Client can be a stand-alone Java application, an applet, servlet or a JSP. This document only covers writing a stand-alone Java application. In order to invoke an enterprise java bean (EJB) that has been deployed into WebSphere on the server side, the Client must do two things: find the EJB on the server, and then create a Client side reference to that EJB. When this Client side reference has been created, the Client can invoke methods upon the EJB as if it was a local object. Clients cannot reference, or invoke, and EJB directly. Any calls made to the EJB must be made through the interfaces that the EJB provides. The interface that is used to create a local reference to the EJB is called the *home interface*. When an EJB is deployed in WebSphere, this home interface is made available to Clients by means of a searchable namespace. This means that a Client can look up an address on the namespace. If there is a home interface at that address, and it is the home interface to the EJB that they were looking for, then the Client can create a local instance of that home interface, and then, from that, a local reference to the EJB can be created.

What code is needed in the Client?

The following code fragment illustrates how to Find and Create a local instance of the Inquiry EJB only. The same must be done to Find and Create a local copy of the Publish EJB.

```
private com.ibm.uddi.ejb.Inquiry inquiry = null;
// This private variable, "inquiry" is going to be the local reference to the EJB in WebSphere
// declaring it outside the scope of a method means that this same reference can be
```

```
// used throughout the client, without having to query the namespace again.
public void homeLookup()
{
 // These variables simply determine the address of the JNDI namespace, and the
  // address of the home interface within that namespace.

 // String naming_factory = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
  //WAS 4.0.2 Naming Factory
 String naming_factory = "com.ibm.websphere.naming.WsnInitialContextFactory";
 //WAS 5.0 Naming Factory

 String namespace_address = "iiop://localhost:2809/";
 //The address of the namespace
 String home_address = "com/ibm/uddi/ejb/InquiryHome";
  //The address of the home interface within the JNDI namespace

 java.util.Hashtable environment = new java.util.Hashtable();
 environment.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, naming_factory);
 environment.put(javax.naming.Context.PROVIDER_URL, namespace_address);

 try
 {
  javax.naming.InitialContext ic = new javax.naming.InitialContext(environment);
 // Create a context using the details above to connect to the namespace

      Object o = ic.lookup(home_address);
       // Do a lookup to see if there is an ejb_home at the address specified above

      // Now create a valid home instance for the EJB type we want to create
  com.ibm.uddi.ejb.InquiryHome home =
       (com.ibm.uddi.ejb.InquiryHome)(javax.rmi.PortableRemoteObject.narrow(o,
          com.ibm.uddi.ejb.InquiryHome.class));

      inquiry = home.create();
   // Now create a local reference of the EJB, by using the home.create() method.
        // Any business method that is intended for the EJB in Websphere
       // must me invoked against this inquiry object.
 }
 catch (javax.naming.NamingException ne) {ne.printStackTrace();}
 // This is thrown if there was a problem connecting to the namespace, or finding
  // the home_address in the namespace
 catch (java.rmi.RemoteException re) {re.printStackTrace();}
 // This usually indicates some sort of system failure, either WebSphere is
  // not running, or there is a communications problem
 catch (javax.ejb.CreateException ce) {ce.printStackTrace();}
 // This is thrown if the EJB reference cannot be created from the home instance.
}
```

Writing Client code to use the EJB API

When the reference to the EJB has been created (the Inquiry Object, in the code shown in the previous paragraph), then the reference can be treated like any other Java object. This is an example method using the UDDI EJB API - the only important point to remember is that, although the Inquiry Object has been created as a local reference, it is still referring to a remote EJB Object in a different server, possibly even in a different country. This means that at the very least a javax.rmi.RemoteException must be caught on each method call that is made to the EJB.

```
public void findBusiness()
{
 System.out.println("Find Business:");
         NameList names = new NameList();
         names.add(new Name("IBM Corporation"));
//Create the list of names to find in the  UDDI Registry, here just one is used, "IBM Corporation"
         try
```

```
        {
            BusinessList list = inquiry.findBusiness(names);
//This is the call to the inquiry EJB that searches through the UDDI Registry

  //Now display the amount of business found, and for each one, get the BusinessKey,
          // the BusinessName and the amount of Services that Business has
             System.out.println("There are "+
               list.getBusinessInfos().size()+
                 " matching Businesses in this registry");
             for (int i=0;i<list getBusinessInfos().size();i++)
        {
                  BusinessInfo business = list.getBusinessInfos().get(i);
                  System.out.println("\nBusinessKey =
                       "+business.getBusinessKey());
                  System.out.println("BusinessName =
                       "+business.getNames().get(0).getNameString());
                  System.out.println("This Business Has "+
                       business.getServiceInfos().size()+
                       " Services\n");
             }
         }

 // This is a UDDI specific exception, and will be thrown if for example an
  //invalid name was used as the search criteria
 catch (com.ibm.uddi.datatypes.DispositionReportException e) {
      this.handleDispositionReportException(e);}

   catch (java.rmi.RemoteException re) {re.printStackTrace();}
 // This is the RemoteException that is thrown if there has been a system
  // failure or a connection problem.
}
```

What new code is needed on the client?

Just as each EJB has an interface listed on the JNDI namespace, the javax.transaction.UserTransaction class also has an interface listed. This means that the same method used to get a local instance of an EJB can be applied to get a local instance of the UserTransaction class. Again, this code can be used to find the UserTransaction reference on the namespace, in addition to the code required to find the Inquiry EJB and the Publish EJB, or, alternatively, there is a slightly more elegant method used in the TransactionEJBClientSample.java.

```
public void txLookup()
{
 private javax.transaction.UserTransaction tx = null;
 // This is the private variable that will be used to hold the UserTransaction Object
  // declaring it outside the scope of a method means that this same reference can be
 // used throughout the client, without having to query the namespace again.


 // These variables simply determine the address of the JNDI namespace,
  // and the address of the home interface within that namespace.

 // String naming_factory = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
//WAS 4.0.2 Naming Factory
 String naming_factory = "com.ibm.websphere.naming.WsnInitialContextFactory";
//WAS 5.0 Naming Factory

 String namespace_address = "iiop://localhost:2809/";
//The address of the namespace
 String transaction_address = "jta/usertransaction";
//The address of the UserTransaction interface within the JNDI namespace

 java.util.Hashtable environment = new java.util.Hashtable();
 environment.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY, naming_factory);
 environment.put(javax.naming.Context.PROVIDER_URL, namespace_address);
```

```
 try
 {
  javax.naming.InitialContext ic = new javax.naming.InitialContext(environment);
// Create a context using the details above to connect to the namespace
      Object remote_object = ic.lookup(transaction_address);
// Do a lookup to see if there is a UserTransaction Object at the address specified above
  tx = (javax.transaction.UserTransaction)remote_object;
// Convert the remote object found into a UserTransaction Object, and assign
// to the private variable
 }
 catch (javax.naming.NamingException ne) {ne.printStackTrace();}
// This is thrown if there was a problem connecting to the namespace, or finding
// the transaction_address in the namespace
}
```

Writing Client code to use the EJB API with a Client transaction

To perform an Inquiry, a Publish or a Delete upon the IBM WebSphere UDDI Registry with client-side transactional support requires very little additional code compared to doing the same operations without client side transactional support. Using the same code that is listed previously (in ″Writing Client Code to use the EJB API″), this example illustrates how easy client side transactions are to implement.

The additional lines of code needed are in bold type. This code also assumes that there is a variable called *tx* that has been declared at the class scope.

```
public void findBusiness()
{
//Just as there are UDDI and RMI specific exceptions thrown,
// 5 more exceptions need to be caught.
try
{
tx.begin(); //This begins the transaction context
System.out.println("Find Business:");
NameList names = new NameList();
names.add(new Name("IBM Corporation"));
//Create the list of names to find in the UDDI Registry, here just one is
//used, "IBM Corporation"
try
{
BusinessList list = inquiry.findBusiness(names);
//This is the call to the inquiry EJB that searches through the UDDI Registry

//Now display the amount of business found, and for each one, get the BusinessKey,
//the BusinessName and the amount of Services that Business has
System.out.println("There are "+list.getBusinessInfos().size()+
  " matching Businesses in this registry");
for (int i=0;i<list.getBusinessInfos().size();i++)
{
BusinessInfo business = list.getBusinessInfos().get(i);
System.out.println("\nBusinessKey = "+business.getBusinessKey());
System.out.println("BusinessName =
   "+business.getNames().get(0).getNameString());
System.out.println("This Business Has "+
  business.getServiceInfos().size()+" Services\n");
}
}
// This is a UDDI specific exception, and will be thrown if for example an
// invalid name was used as the search criteria
catch (com.ibm.uddi.datatypes.DispositionReportException e) {
  this.handleDispositionReportException(e);}
catch (java.rmi.RemoteException re) {re.printStackTrace();}
```

```
// This is the RemoteException that is thrown if there has been a system
// failure or a connection problem.

tx.commit(); //This ends the transaction context
}
catch (javax.transaction.NotSupportedException nse) {
  nse.printStackTrace();}
catch (javax.transaction.RollbackException rbe) {
  rbe.printStackTrace();}
catch (javax.transaction.SystemException se) {
  se.printStackTrace();}
catch (javax.transaction.HeuristicMixedException hme) {
  hme.printStackTrace();}
catch (javax.transaction.HeuristicRollbackException hrbe) {
  hrbe.printStackTrace();}
}
```

- The datatypes package
- Methods in the EJB Interface

Datatypes package in the UDDI Registry

The following table lists the classes in the com.ibm.uddi.datatypes package, the elements in the UDDI v2.0 XML schema, and the correspondence between the two.

com.ibm.uddi.datatypes Class	Corresponding UDDIv2.0 XML Schema Element	Notes on DatatypeClass
AccessPoint	accessPoint	
Address	address	
String	addressLine	
AdressLineList		Encapsulates a vector of addressLine Strings
AddressList		Encapsulates a vector of Address objects
AssertionStatusItem	assertionStatusItem	
AssertionStatusItemList		Encapsulates a vector of AssertionStatusItem objects
AssertionStatusReport	assertionStatusReport (response message)	
String	authInfo	
AuthToken		Object containing authInfo String and operator name
String	bindingKey	
BindingDetail	bindingDetail (response message)	
BindingTemplate	bindingTemplate	
BindingTemplateList	bindingTemplates	Encapsulates a vector of Bindingtemplate objects
BusinessDetail	businessDetail (response message)	
BusinessDetailExt	businessDetailExt (Response message)	**
BusinessEntity	businessEntity	
BusinessEntityExt	businessEntityExt	**
BusinessEntityExtList		Encapsulates a vector of BusinessEntityExt objects **
BusinessEntityList		Encapsulates a vector of BusinessEntity objects
BusinessInfo	businessInfo	

BusinessInfoList	businessInfo	Encapsulates a vector of businessInfo objects
String	businessKey	
BusinessList	businessList (response message)	
BusinessService	businessService	
BusinessServiceList	businessServices	Encapsulates a Vector of BusinessService objects
CategoryBag	categoryBag	
String	completionStatus	
Contact	contact	
ContactList	contacts	Encapsulates a vector of Contact objects
Description	description	
DescriptionList		Encapsulates a vector of Description objects
DiscoveryUrl	discoveryURL	
DiscoveryUrlList	discoveryURLs	Encapsulates a vector of DiscoveryURL objects
DispositionReport	dispositionReport	
DispositionreportException		Exception thrown by EJB interface functions when an error occurs
Email	email	
EmailList		Encapsulates a vector of Email objects
EndPoint		Used as baseclass for AccessPoint and HostingRedirector providing mutual exclusivity
ErrInfo	errInfo	
	findQualifier	
FindQualifier	findQualifiers	
String	fromKey	
HostingRedirector	hostingRedirector	
IdentifierBag	identifierbag	
InquiryOptions		Encapsulates a FindQualifiers object and a maxrows field. Used in find_* API calls to specify search options
InstanceDetails	instanceDetails	
String	instanceParms	
String	keyValue	
KeyedReference	keyedReference	
keysOwned	keysOwned	
LanguageString		Abstract class, extended by some of the datatypes, which represents a string that can optionally be tagged with xml:lang.
Name	name	
NameList		Encapsulates a vector of Name objects
OverviewDoc	overviewDoc	
String	overviewURL	
String	personName	
Phone	phone	

PhoneList		Encapsulates a vector of Phone objects
PublisherAssertion	publisherAssertion	
PublisherAssertionList		Encapsulates a vector of Publisher Assertion objects
PublisherAssertions	publisherAssertions (response message)	
RegisteredInfo	registeredInfo (response message)	
	relatedBusinessInfo	Not used
	relatedBusinessInfos	Not used
RelatesBusinessesList	relatedBusinessesList	
RelatedBusinessInfo	relatedBusinessInfo	
RelatedBusinessInfos	relatedBusinessInfos	
Result	result	
ResultList		Encapsulates a Vector of Result objects
ServiceDetail	serviceDetail (response message)	
ServiceInfo	serviceInfo	
ServiceInfoList	serviceInfos	Encapsulates a vector of serviceInfo objects
String	serviceKey	
ServiceList	serviceList (response message)	
	sharedRelationships	Not used
SharedRelationships	sharedRelationships	
Tmodel	tModel	
TModelBag	tModelBag	
TModelDetail	tModelDetail (response message)	
TModelInfo	tModelInfo	
TModelInfoList	tModelInfos	Encapsulates a vector of TModelInfo objects
TModelInstanceInfo	tModelInstanceInfo	
TModelInstanceInfoList	tModelInstanceDetails	Encapsulates a vector of TModelInstanceInfo objects
String	tModelKey	
TModelList	tModelList (response message)	
TModels		Encapsulates a vector of TModel objects
String	toKey	
String	uploadRegister	
UploadRegisterList		Encapsulates a vector of uploadRegister strings

** Used in UDDI API functions relating to BusinessDetailExtension. These UDDI API functions are not implemented in Version 1 of the IBM WebSphere UDDI Registry.

In general, a datatype called DatatypeList contains a vector of *Datatype* objects. Often these correspond to XML schema elements with plural names. (For example the datatype *Contact* corresponds to XML element *contact*, and *ContactList*

corresponds to *contacts*.) Where there is no "plural" XML schema element for a particular *Datatype*, often there is still a *DatatypeList* where it is useful to have one, for example *AddressList*.

The exceptions to this naming convention occur when there is an existing XML schema element ending in "List". The exceptions are: TModelList, ServiceList, BusinessList. In these cases, the corresponding datatypes are given the same names as the XML schema elements, and the datatypes that would have had these names are called: TModels, BusinessServiceList, BusinessEntityList.

EJB interface methods in the UDDI Registry

Inquiry

```
findBinding
findBusiness
findRelatedBusinesses
findService
findTModel
getBindingDetail
getBusinessDetail
getServiceDetail
getTModelDetail
```

Publish

```
addPublisherAssertions
deleteBinding
deleteBusiness
deletePublisherAssertions
deleteService
deleteTModel
getAssertionStatusReport
getRegisteredInfo
getPublisherASsertions
saveBinding
saveBusiness
saveService
saveTModel
setPublisherAssertions
```

Each method is overloaded and can take various combinations of arguments. The Javadoc information contains detailed information about each method.

Note that the get_authToken and discard_authtoken methods are not implemented, as WebSphere security is used instead.

UDDI troubleshooting tips

When the IBM WebSphere UDDI Registry is running, it might issue messages to report events or errors. You can use these messages, described in Messages as your first aid to problem determination. If you need more details about the causes of a problem, you can turn on tracing for UDDI, as described in:

- Turning on UDDI trace
- **Common causes of errors**

  Below are a few of the common causes of errors that might be found and their suggested solutions.

  - If you set up the UDDI Registry application with a JDBC driver and datasource that reference Cloudscape, but set the persister property in *uddi.properties* to specify DB2, **or vice versa**, then some unexpected behavior

results, such as a fatal error on deleting an entity. If this happens, you should check that the above details are not in conflict. This only applies to a UDDI Registry installation on a single appserver.

– If you get a message "The application failed to initialize" when trying to access the UDDI User console and you are using DB2 as the persistence store for the UDDI Registry, a likely cause of the problem is that you specified the wrong userid and/or password when you ran the script to install the UDDI Registry application. If this occurs rerun the script ensuring you use the correct userid and password.

  Alternatively, on Unix platforms, you may not have run the db2profile before deploying the UDDIReg application or before starting the application server.

– You might find that, after uninstalling and reinstalling the UDDI Registry, you get errors from the UDDI User Console of the form:

```
"Error 500: JSPG0059E: Unable to compile class for JSP".
```

  If this occurs, then you should clear out the temp directory of the WebSphere AppServer.

– When running one of the UDDI setup scripts setupuddi.jacl or removeuddi.jacl, if you get an error such as:

```
WASX7017E: Exception received while running file "setupuddi.jacl";
 exception information: com.ibm.bsf.BSFException: error while eval'ing Jacl
 expression: java.util.MissingResourceException:
   Can't find resource for bundle java.util.PropertyResourceBundle,
      key ErrMsgIncorrectNumArgs
```

  ensure that the file setupuddimessages.jar is located in the *lib* subdirectory of the WebSphere deployment manager or application server under which you are running the script.

– When running the DB2 Setup Wizard, if you get an error stating "Invalid userid and password", it could be caused by any of the following situations:
  - You have supplied an invalid userid or password - re-enter with a valid userid and password.
  - The supplied userid does not have the necessary privileges - retry with a userid that has appropriate privileges.
  - DB2 is stopped when you run the wizard - start DB2 and retry the wizard.
  - The UDDI20 database already exists and has been removed previously and, as such, is not catalogued. The DB2 wizard does not recognize this situation and gives the error. You now have two options.
    1. To use the existing database, catalogue it, and there is no need to rerun the Wizard.
    2. To create a new database, recatalog the database and re-run the DB2 wizard and choose the option to overwrite the database. (Any existing data **WILL** be lost.)

  5.0.1 +

  • Catalog the database by:
    – **On Windows:**

```
>db2cmd
>db2cat -d uddi20
```

    – **On Unix platforms**

```
>su db2inst1 (or name of your db2 instance)
     >db2 CATALOG DATABASE UDDI20
```

  - 5.0.1 + ensure that, if you are using a non-English installation of DB2, you have applied PTF501.

Note: 5.0.1 + If you have a copy of the file SetupDB2UDDI.jar in your appserver directory, the application of the base and Network Deployment PTFs will not update SetupDB2UDDI.jar in your appserver directory. You must apply the PTF for Network Deployment to your DeploymentManager file structure to update the SetupDB2UDDI.jar located there (in the /IDDIReg/scripts subdirectory), and then manually copy this jar file to any application server you may wish to run it on.

– There is a limitation concerning URL rewriting causing JavaScript syntax errors on several Web pages in the UDDI User Console. Because of this, cookies must be enabled in client browsers, the application server must have cookies enabled as the session tracking mechanism, and URL rewriting must be disabled.
– If you have an existing DB2 version of the UDDI Registry database, and you use the UDDI DB2 setup wizard to replace this database with a new one, and if the database is in use at the time that you run the UDDI DB2 setup wizard, then the existing database is not overwritten.
– When running the UDDI DB2 setup wizard, as part of the installation step ″Setting up the UDDI Registry to use DB2 within a deployment manager cell″ or ″Setting up the UDDI Registry to use DB2 in a single application server″, in addition to running the *was_install\bin\setupcmdline.bat* directory, you should also enter either ***set PATH=%WAS_PATH%*** (for Windows platforms) or ***export PATH=/opt/WebSphere/AppServer/java/bin:$PATH*** (for Unix platforms) to ensure that you have access to Java.
– UDDI user console ″Page cannot be displayed″ errors with Internet Explorer.

  If you use Internet Explorer with the option ″Show friendly HTTP error messages″ enabled and you have WebSphere Application Server security enabled (user ID and password authentication enabled), you might experience intermittent errors on the browser, such as ″Page cannot be displayed″, when navigating the UDDI user console. This might be particularly noticeable when accessing the publish actions.

  To avoid such errors, disable the ″Show friendly HTTP error messages″ option on Internet Explorer. This option is found under **Tools > Internet options > Advanced Tab > Browsing Section**
– When using SOAP or UDDI4J, it is sometimes necessary to call setServiceKey (″″) before saving your changes, except with the EJB interface where this might result in an error.
– There are known problems with inquiries issued against the UDDI Registry if IBM Cloudscape is used as the persistence store for the registry data. Certain complex inquiries might produce unexpected results, or could fail. If your application needs to make inquiries of this nature, consider using DB2 as the persistence store. Note: DB2 must be used for production purposes. The IBM Cloudscape support is only provided for development and test use.
– You might see errors if you specify requests that specify more than 5 category values, more than 5 identifier values, or more than 5 technical model (tModel) values.
– If you stop and restart the UDDI Registry application from the administrative console, and then try to access the Registry through the user console, an *″Error 500 - object is not an instance of a declaring class″* displays on the user console, and the error message *″SRVE0026E″* displays in the system log. You cannot access the UDDI Registry until you restart the WebSphere Application Server. To avoid this problem when restarting the UDDI Registry, you should set ***Prefer WEB-INF classes*** on the panel navigated to by the following steps: **Applications > Manage Applications > UDDIRegistry > Web Modules defined for this Application > gui.war**.

– It is possible that a scripting error displays when you are running the **wsadmin appserversetupuddi.jacl** command.

  During installation, if you see the following error at the end of running the **appserversetupuddi.jacl** command, you can safely ignore the error. It is recommended that you start (or stop and restart) the application server and then continue.

  Here is an example of the error:

```
UDIN2041I: Starting UDDI application. UDIN8019E: startApplication command for
   appname caught exception Exc. Values are: appname=UDDIRegistry,
   Exc=com.ibm.ws.scripting.Scripting
   Exception: com.ibm.websphere.management.exception.Connector
   Exception: ADMC0009E: Failed to make the SOAP RPC call: invoke
```

– If you find that the UDDI user console is giving unexpected results, in particular if you have installed a FixPack but are not seeing all of the fixes, then you should clear out the compiled UDDI JSP files from the temp directory of the WebSphere Application Server. These files will be in a directory identified by the name of the UDDI Registry located beneath the temp directory.

– If attempting to use a remote DB2 database and you are experiencing problems attaching to the remote system, one of the possible causes might be IP addressing. You should not have this problem if the remote system is using a static IP address. If, however, the remote system is using DHCP, the two systems must be aware of each others subnet mask.

  For Windows, the subnet mask can be found by starting a Command Prompt and entering *"ipconfig"* on the remote system. On the host system, the WINS might need to be edited to add the remote subnet mask. To do this on Windows go to the following commands:

  1. START => Network and Dial-up Connections => Local Area Network Connection 2 => Internet Protocol (TCP/IP) and click on Properties
  2. Click on *"Advanced"*.
  3. Click on the WINS tab and add the new subnet mask
  4. Move the new subnet mask to the top of the list by highlighting it and pressing the *"Up"* arrow until it is the top of the list of WINS addresses

  On Unix platforms, you can use *ifconfig* to determine the subnet mask.

- **Known limitations with UDDI Utility Tools and workarounds**

  There are known limitations with the UDDI Utility Tools and a workaround for each:

  – Referenced businesses in service projections are not added automatically to the EDF in the same manner as referenced tModels.

    Workaround: Add the referenced business that will 'own' the projected service to the EDF. If the business is not present in the target registry, it should be placed before the service's owning business in the EDF.

  – Cycle detection for service projections are not detected in the same manner as for referenced tModels.

    Workaround: If a circular reference is present between two or more service projections, break the cycle by removing one of the projections temporarily, perform the import and update the changed entity to reestablish the cycle in the target registry.

  – tModels that were deleted (in the logical sense) in the source registry are imported and promoted as undeleted in the target registry. This is because, in the UDDI Version 2 specification, the deleted state of tModels is not exposed as API calls.

Workaround: After importing the tModel, perform a delete. This is done using the UDDI Utility Tools delete function, or any other UDDI Registry API access method.

– BindingTemplates referenced by hostingRedirectors are not added automatically to the EDF in the same manner as referenced tModels.

  Workaround: Add the referenced bindingTemplate to the EDF.

– Businesses referenced by an 'owningBusiness' keyedReference are not automatically added to the EDF.

  Workaround: Import the referenced business into the target registry before importing the tModel that references it.

– The JSSE provider class, when security is enabled, is not configurable. It must be com.ibm.jsse.IBMJSSEProvider.

– A few combinations of command line arguments are not validated and prevented, for example, it is possible to specify -import with -keysFile <path to file> in the same command, although the ‑keysFile is ignored.

Turning on UDDI trace

You enable UDDI-specific trace in the same way as you enable other tracing in the WebSphere Application Server.

The following is a list of trace strings that you can use:

- com.ibm.uddi.api
- com.ibm.uddi.config
- com.ibm.uddi.datatypes
- com.ibm.uddi.dom
- com.ibm.uddi.ejb
- com.ibm.uddi.exception
- com.ibm.uddi.exceptions
- com.ibm.uddi.gui
- com.ibm.uddi.gui.inquire
- com.ibm.uddi.gui.publish
- com.ibm.uddi.persistence
- com.ibm.uddi.persistence.jdbc
- com.ibm.uddi.persistence.jdbc.cloudscape
- com.ibm.uddi.persistence.jdbc.db2
- com.ibm.uddi.ras
- com.ibm.uddi.security
- com.ibm.uddi.soap
- com.ibm.uddi.uuid
- com.ibm.uddi.validation
- com.ibm.uddi.xml

For example, to trace the UDDI user console:

```
'com.ibm.uddi.gui=all=enabled'
```

This enables all types of trace for the gui.

Messages

When the IBM WebSphere UDDI Registry is running, it issues messages to report events or errors. The messages are in the form `UDxxnnnns` where:

xx is a two character descriptor identifying which component is involved

nnnn is the error code

s is either I (Information) or E (Error)

The prefix *UDxxnnnns:* is followed by text that describes the event or error. For some messages, the first word of the text is one of the form (MSN=SSSS). The SSSS provides a message sequence number (or MSN), which identifies the unique circumstance in which the message was issued, and is of use where the same message can be issued in more than one circumstance.

To help you diagnose problems and minimize the need to enable trace in any of the above components, view the messages table. You can view the messages by prefix or component, whichever is easiest for you to find in the table. All messages are documented with user/system action and explanation.

The text for the UDDI messages is contained in a file *uddiresourcebundles.jar* which is placed, by the installation process, into the \lib subdirectory (Windows) of the WebSphere application server into which the UDDI Registry was installed. If you will be running a console or log analyzer from another process; for example, to analyze the activity log, you must place a copy of *uddiresourcebundles.jar* into a directory that is within the classpath of that process. Otherwise, the message lookup for the UDDI messages will fail.

UDDI Components Message Prefix Table **Click on individual links for message documentation for the component**	
UDAI	API
UDCF	Configuration
UDDA	Datatypes
UDDM	DOM
UDEJ	EJB Interface
UDEX	Exceptions
UDIN	Installation
UDLC	Local API
UDPR	Persistence
UDRS	Logging
UDSC	Security
UDSP	SOAP Interface
UDUC	User Console
UDUU	UUID

UDAI (Web Services UDDI) messages

There are no messages issued by this component.

UDCF (Web Services UDDI) messages

UDCF0001E: Exception occurred while getting int value of configuration property ″<property>″, exception: ″<exception>″

Explanation: This message is issued when an attempt to read the value of a configuration property from the uddi.properties file and convert it to integer has failed with the indicated exception.

User Response: Check that the uddi.properties file contains a value for the indicated configuration property, and that the value is valid. Check also

that the indicated configuration property is a legal property. Refer to the InfoCenter for further information about global configuration properties and the uddi.properties file.

UDCF0002E: Exception occurred while getting long value of configuration property ″<property>″, exception: ″<exception>″

Explanation: This message is issued when an attempt to read the value of a configuration property from the uddi.properties file and convert it to long has failed with the indicated exception.

User Response: Check that the uddi.properties file contains a value for the indicated configuration property, and that the value is valid. Check also that the indicated configuration property is a legal property. Refer to the InfoCenter for further information about global configuration properties and the uddi.properties file.

UDCF0003E: Exception occurred while getting boolean value of configuration property ″<property>″, exception: ″<exception>″

Explanation: This message is issued when an attempt to read the value of a configuration property from the uddi.properties file and convert it to boolean has failed with the indicated exception

User Response: Check that the uddi.properties file contains a value for the indicated configuration property, and that the value is valid. Check also that the indicated configuration property is a legal property. Refer to the InfoCenter for further information about global configuration properties and the uddi.properties file.

UDCF0004E: Failed to load UDDI global properties file.

Explanation: This message is issued when the UDDI global configuration properties file, uddi.properties, cannot be loaded. Default values for the global configuration properties will be set, but these defaults may not be adequate for many of the properties, so you should investigate and resolve this problem.

User Response: Check that the uddi.properties file exists and is in the correct directory. Refer to the InfoCenter for further information about global configuration properties and the uddi.properties file.

UDCF0005E: Exception occurred while loading UDDI global configuration properties, exception: ″<exception>″

Explanation: This message is issued when an attempt to load the UDDI global configuration properties from the uddi.properties has failed with the indicated exception. Default values for the global configuration properties will be set, but these defaults may not be adequate for many of the properties, so you should investigate and resolve this problem.

User Response: Check that the uddi.properties file exists and contains valid values for each of the configuration properties. Refer to the InfoCenter for further information about global configuration properties and the uddi.properties file.

UDDA (Web Services UDDI) messages

There are no messages issued by this component.

UDDM (Web Services UDDI) messages

There are no messages issued by this component.

UDEJ (Web Services UDDI) messages

There are no messages issued by this component.

UDEX (Web Services UDDI) messages

There are no messages issued by this component.

UDIN (Web Services UDDI) messages

UDIN0001I: Assuming hard coded defaults.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None..

UDIN0002I: Cloudscape classpath is clpath. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0003I: Looking for childtype childname under parenttype parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0004I: Looking for childtype childname under parenttype parentname and parenttype2 parentname2. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0005I: Conflict found with existing childtype childname. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0006I: Not creating requested childtype. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0007I: Seeking parenttype with requested id of parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0008I: Seeking parenttype with requested id of parentname under parenttype2 parentname2. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0009I: Attempting to create childtype under parenttype of parentID. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0010I: Create command that will be issued is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0011I: childtype childId was successfully created. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0012I: Looking for builtin_rra.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0013I: List for J2CResourceAdapter returned N members. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0014I: Hunting J2CResourceAdapter associated with Node nodename. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0015I: Using rraID as builtin_rra. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0016I: Using provider class of implclass with a classpath of clpath. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0017I: Installing to server servername, node nodename using database type of dbtype. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0018I: Attempting to create UDDI JDBCProvider.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0019I: Attempting to create UDDI Datasource.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0020I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0021I: Attempting to install UDDI Registry application.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0022I: Checking for installed UDDI Registry application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0023W: Application of name appname is not present. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0024I: ApplicationManager not running, so application will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0025I: Stopping application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0026W: stopApplication command for application appname caught exception Exc. Application might not have been running on this server. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0027I: Application appname stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0028I: Removing application appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0029I: Application appname removed successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0030I: Adding resource bundles to repository.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0031I: Adding Cloudscape user functions to repository.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0032I: UDDI configuration properties file already exists. Only the persister and getServletURLPrefix properties will be overwritten.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0033I: Editing UDDI configuration properties file propsfile. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0034I: Url prefix found. Updating it to discoveryURL. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0035I: Persister property found. Updating it to dbtype. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0036I: Adding UDDI configuration properties file to repository for cell cellname under target node and server. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0037I: ws.ext.dir processing starting.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0038I: serverID is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0039I: JVM is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0040I: Out of N properties we located M matches at positions poslist. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0041I: Building new ws.ext.dirs properties.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0042I: SYSPROP is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0043I: ws.ext.dir has been set with new sysprop. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0044I: ws.ext.dir update skipped, required changes already present.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0045I: ws.ext.dir processing step complete.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0046I: Cleaning up temporary version of properties file temppropsfile. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0047I: Issuing nodeSync.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0048I: UDDI Registry successfully installed. Please restart server servername to activate configuration changes. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0049I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0050I: Server is not running, so will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0051I: Stopping server servername under node nodename. Values are:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0052I: Server servername stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0053I: Restarting application server

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0054I: Application server servername restarted successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0055I: Please ignore any errors concerning the serverStartupSyncEnabled attribute.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0101I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0102I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0103I: Changes were not saved on this call.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0104I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0105I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0106I: Attempting to save ws.ext.dir changes.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0107I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0108I: Attempting final save of new configuration.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN0109I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script setupuddi.jacl.

User Response: None.

UDIN1001I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1002I: Server is not running, so will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1003I: Stopping server servername under node nodename. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1004I: Server servername stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1005I: Resource bundles file will be removed from repository if present.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1006I: Removing resource bundles from repository.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1007I: Resource bundles successfully removed from repository.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1008I: Cloudscape user functions file will be removed from repository if present.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1009I: Removing Cloudscape user functions from repository.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1010I: Cloudscape user functions successfully removed from repository.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1011I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1012I: Checking for installed UDDI Registry application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1013W: Application of name appname is not present. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1014I: ApplicationManager not running, so application will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1015I: Stopping application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1016W: stopApplication command for application appname caught exception Exc. Application might not have been running on this server. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1017I: Application appname stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1018I: Removing application appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1019I: Application appname removed successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1020I: UDDI datasource will be removed from server servername in node nodename if present. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1021I: Removing UDDI datasource.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1022I: UDDI datasource successfully removed.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1023I: UDDI JDBC driver will be removed from server servername in node nodename if present. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1024I: Removing UDDI JDBC driver.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1025I: UDDI JDBC driver successfully removed.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1026I: UDDI configuration properties file will be removed from repository if present.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1027I: Removing configuration properties file from cell cellname, node nodename and server servername. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1028I: Configuration properties file successfully removed from repository.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1029I: Issuing nodeSync.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1030I: UDDI Registry application, JDBC driver and datasource removed successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1031I: Restarting application server.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1032I: Application server servername restarted successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1033I: Please ignore any errors concerning the serverStartupSyncEnabled attribute.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1034I: ws.ext.dir processing starting.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1035I: serverID is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1036I: JVM is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1037I: Out of N properties we located M matches at positions poslist. Values are:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1038I: Removing UDDI values from ws.ext.dirs properties.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1039I: ws.ext.dir has been set with new sysprop. Value is:

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1040I: ws.ext.dir update skipped, required changes already present.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1041I: ws.ext.dir processing step complete.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1101I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1102I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1103I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1104I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1105I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1106I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1107I: Attempting final save of new configuration.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1108I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1109I: Attempting to save ws.ext.dir changes.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN1110I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script removeuddi.jacl.

User Response: None.

UDIN2001I: Assuming hard coded defaults.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2002I: Listing members of type parenttype. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2003I: List for type parenttype returned N members. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2004I: Seeking parenttype with requested id of parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2005I: Checking parentID with parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2006I: Using this as parenttype of parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2007I: Checking for existing childtype under parentname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2008I: List for childtype returned N members. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2009I: No existing childtype present. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2010I: N existing objects of type childtype found, examining for conflict with childname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2011I: Checking childID with name childname. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2012I: Conflict found with existing childtype of id childID. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2013I: Not creating requested object of type childtype. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2014I: Conflict found with existing childtype, removing existing childtype. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2015I: Removal of childtype was successful. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2016I: Not in conflict.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2017I: Attempting to create childtype under parentname of parentID. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2018I: Create command that will be issued is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2019I: childtype childID was successfully created. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2020I: No matches found.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2021I: Looking for builtin_rra.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2022I: List for J2CResourceAdapter returned N members. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2023I: Hunting J2CResourceAdapter associated with Node nodename. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2024I: Using rraID as builtin_rra. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2025I: Using provider class of implclass with a classpath of clpath. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2026I: Installing to node nodename using database type of dbtype. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2027I: Attempting to create UDDI JDBCProvider.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2028I: Attempting to create UDDI Datasource.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2029I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2030I: Attempting to install UDDI Registry application.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2031I: Checking for installed UDDI Registry application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2032I: List for Applications returned N members. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2033W: Application of name appname is not present. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2034I: ApplicationManager not running, so application will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2035I: Stopping application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2036W: stopApplication command for application appname caught exception Exc. Application might not have been running on this server. Values

are: **Explanation:** This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2037I: Application appname stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2038I: Removing application appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2039I: Application appname removed successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2040I: Attempting to install application appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2041I: Starting UDDI application.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2042I: Application appname started successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2101I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2102I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2103I: Changes were not saved on this call.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2104I: Attempting to save post installation configuration.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2105I: Changes saved successfully for UDDI Registry.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2106I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN2107I: Changes saved successfully for UDDI Registry.

Explanation: This is an informational message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: None.

UDIN3001I: Application Manager appmgr found. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3002I: Checking for installed UDDI Registry application.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3003I: List for Applications returned N members. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3004W: Application of name appname is not present. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3005I: ApplicationManager not running, so application will not need to be stopped.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3006I: Stopping application of name appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3007W: stopApplication command for application appname caught exception Exc. Application might not have been running on this server. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3008I: Application appname stopped successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3009I: Removing application appname. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3010I: Application appname removed successfully. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3011I: UDDI datasource will be removed from server servername in node nodename if present. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3012I: Removing UDDI datasource.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3013I: UDDI datasource successfully removed.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3014I: UDDI JDBC driver will be removed from server servername in node nodename if present. Values are:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3015I: Removing UDDI JDBC driver from node nodename. Value is:

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3016I: UDDI JDBC driver successfully removed.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3017I: UDDI Registry application, JDBC driver and datasource removed successfully.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3101I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3102I: Changes to remove UDDI Registry have been saved successfully.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3103I: Attempting to save new configuration.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3104I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3105I: Attempting final save of new configuration.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN3106I: Changes saved successfully.

Explanation: This is an informational message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: None.

UDIN6001E: This script must be run in a Deployment Manager environment.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6002E: To install in a standalone application server, use appserversetupuddi.jacl.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6003E: Incorrect number of arguments passed to script.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6004E: Usage is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl. The text following 'Usage is:' gives the syntax for calling setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6005E: (<db2userid> <db2password> <db2ziplocation> are only required if setting up to use DB2).

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6006E: Use all forward ('/') slashes to avoid problems with escaping back ('\\') slashes.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6007E: Removal of childtype childname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6008E: An exception Exc occurred while creating childtype. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6009E: Unable to find requested parentype of parentname. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6010E: List command for J2CResourceAdapter caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6011E: No J2CResourceAdapter objects available.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6012E: An error occurred during execution of setupuddi.jacl. Please check the parameters and try again.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6013E: Uninstall of application appname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6014E: Install of UDDI application caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6015E: Could not get JVM.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6016E: Cannot find nodeSync MBean.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6017E: nodeSync failed. UDDI Application may not be fully installed.

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6018E: stopServer command for server servername caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6019E: startServer command for server servername caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6101E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6102E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6103E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN6104E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script setupuddi.jacl.

User Response: This message is self-explanatory.

UDIN7001E: This script must be run in a Deployment Manager environment.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7002E: To remove from a standalone application server, use appserverremoveuddi.jacl.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7003E: Incorrect number of arguments passed to script.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7004E: Usage is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl. The text following 'Usage is:' gives the syntax for calling removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7005E: stopServer command for server servername caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7006E: Removal of resource bundles caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7007E: Removal of Cloudscape user functions caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7008E: Uninstall of application appname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7009E: Removal of UDDI datasource caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7010E: Removal of UDDI JDBC driver caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7011E: Removal of configuration properties file caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7012E: Cannot find nodeSync MBean.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7013E: nodeSync failed. UDDI Application may not be fully uninstalled.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7014E: startServer command for server servername caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7015E: Could not get JVM.

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7101E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7102E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7103E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7104E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN7105E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script removeuddi.jacl.

User Response: This message is self-explanatory.

UDIN8001E: This script must be run on a standalone application server.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8002E: To install in a Deployment Manager Environment, use setupuddi.jacl.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8003E: Incorrect number of arguments passed to script.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8004E: Usage is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl. The text following 'Usage is:' gives the syntax for calling appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8005E: (<db2userid> <db2password> <db2ziplocation> are only required if setting up to use DB2).

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8006E: Use all forward ('/') slashes to avoid problems with escaping back ('\\') slashes.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8007E: List command for type parenttype caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8008E: No objects of type parenttype available. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8009E: List command for childtype caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8001E: This script must be run on a standalone application server.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8010E: Error during remove of existing childtype, exception Exc caught. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8011E: Create command for childtype caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8012E: Unable to find requested parentype of parentname. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8013E: List command for J2CResourceAdapter caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8014E: No J2CResourceAdapter objects available.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8015E: An error occurred during execution of appserversetupuddi.jacl. Please check the parameters and try again.

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8016E: List command for Applications caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8017E: Uninstall of application appname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8018E: Install of UDDI application caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8019E: startApplication command for appname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8101E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8102E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN8103E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserversetupuddi.jacl.

User Response: This message is self-explanatory.

UDIN9001E: This script must be run on a standalone application server.

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9002E: To remove from a deployment manager environment, use removeuddi.jacl.

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9003E: Incorrect number of arguments passed to script.

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9004E: Usage is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9005E: List command for Applications caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9006E: Uninstall of application appname caught exception Exc. Values are:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9007E: Removal of UDDI datasource caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9008E: Removal of UDDI JDBC driver caught exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9101E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9102E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDIN9103E: Error saving configuration, changes not saved due to exception Exc. Value is:

Explanation: This is an error message issued by the UDDI setup script appserverremoveuddi.jacl.

User Response: This message is self-explanatory.

UDLC (Web Services UDDI) messages

There are no messages issued by this component.

UDPR (Web Services UDDI) messages

There are no messages issued by this component.

UDRS (Web Services UDDI) messages

UDRS0001E: Exception ″<exception>″ occurred while attempting to get UDDI Message Logger.

Explanation: This message is issued to stderr when an attempt to get the UDDI Message Logger fails with the indicated exception. Since the attempt to get the message logger failed, the message cannot be logged. No messages can be logged by this instance of the IBM WebSphere UDDI Registry.

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDRS0002E: Exception ″<exception>″ occurred while attempting to get UDDI Trace Logger for ″<component>″.

Explanation: This message is logged when an attempt to get the UDDI Trace Logger for the specified component (or package) fails with the indicated exception. No trace entries can be logged for this component or package of the IBM WebSphere UDDI Registry.

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSC (Web Services UDDI) messages

There are no messages issued by this component.

UDSP (Web Services UDDI) messages

UDSP0001E: ParserPool found empty whilst attempting to process request. Request unsatisfied

Explanation: A SOAP request was received, but was unable to be dealt with, as there were no free Parsers within the ParserPool.

User Response: Consider increasing the number of Parsers within the ParserPool by modifying the Init Parameter on the SOAP servlets.

UDSP0002E: Error locating schemas required for UDDI processing. SOAP Servlets unworkable.

Explanation: The SOAP servlet was unable to locate the schemas it requires in order to process SOAP requests. Without these, the servlet cannot process SOAP requests.

User Response: Check installation of UDDI was performed correctly. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSP0003W: Servlet unable to locate init parameter 'defaultPoolSize'. Using internal defaults.

Explanation: The SOAP servlet was unable to locate the init parameter which sets the default size of the ParserPool. It will fall back to an internal default.

User Response: If this message occured after attempting to make changes to the defaultPoolSize init parameter, ensure the changes were correct. If this message has appeared after installed, ensure installation was performed correctly.

UDSP0004W: Servlet unable to understand init parameter 'defaultPoolSize'. Using internal defaults.

Explanation: The SOAP servlet was unable to parse the init parameter which sets the default size of the ParserPool. It will fall back to an internal default.

User Response: If this message occured after attempting to make changes to the defaultPoolSize init parameter, ensure the changes were correct. If this message has appeared after installed, ensure installation was performed correctly.

UDSP0005E: Error occurred during parser creation.

Explanation: An unspecified error occured during the creation of a SOAP parser

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSP0006E: Internal configuration error.

Explanation: This error may occur if there was a failure creating a Parser, with accompanying message UDSP0005. It may also occur if there was a problem acquiring the Persistence layer.

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSP0007E: Error during servlet acquisition of persistence layer.

Explanation: The SOAP servlet was unable to acquire the persistence layer required for it to communicate with the UDDI datasource

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSP0008E: Error during servlet release of persistence layer.

Explanation: The persistence layer reported a problem when the SOAP servlet attempted to release it.

User Response: Restart the UDDI registry. If the error persists, examine the WebSphere logs for information on its cause. If the problem cannot be resolved, then please contact the IBM Customer Service Center.

UDSP0009E: Error during sending of response to client.

Explanation: An error occured when sending a SOAP response message back to a client. The client may not have received the response

User Response: This error is recorded to enable logging of failed responses to clients. The error may be the fault of the client disconnecting before the

reply could be sent, or may indicate a network problem. Examine the WebSphere logs for more information on its cause.

UDUC (Web Services UDDI) messages

UDUC0001I: IBM WebSphere UDDI Registry user console starting initialization.

Explanation: The user console control servlet is starting.

User Response: None.

UDUC0002I: IBM WebSphere UDDI Registry user console finished initialization.

Explanation: The user console control servlet has completed startup.

User Response: None.

UDUC0003I: Reading init parameters.

Explanation: The user console control servlet has started reading external parameters in its init method

User Response: None.

UDUC0004I: Finished reading init parameters.

Explanation: The user console control servlet has finished reading external parameters in its init method. This message indicates the user console is ready to accept client requests.

User Response: None.

UDUC0005E: A serious error has occurred. Error message: <Message> error: <Throwable>. More information: <Additional information>.

Explanation: This error message indicates an unexpected error has occurred. The <Message> describes the error that has occurred and the <Throwable> is the type of error that was caught. <Additional information> may provide further information, if available.

User Response: A trace of the gui component is recommended. Contact IBM support with this information.

UDUC0006E: A persistence error has occurred. Error message: <Message> error: <Throwable>. More information: <Additional information>.

Explanation: An error occurred while performing a database operation. The <Message> describes the error that occurred and the <Throwable> is the type of error that was caught. <Additional information> may provide further information, if available.

User Response: Check database connections and state. Please provide IBM support with a trace, including the gui and persistence components.

UDUC0007E: A User mismatch error has occurred. Error message: <Message> error: <Throwable>. More information: <Additional information>.

Explanation: The user id provided does not match the user id required or expected whilst performing an operation that requires authentication. The <Message> describes the error that occurred and the <Throwable> is the type of error that was caught. <Additional information> may provide further information, if available.

User Response: Check the user has authority for the operation being requested. If necessary, contact IBM support detailing the actions taken to recreate the problem.

UDUC0008E: An invalid key was passed. Error message: <Message> error: <Throwable>. More information: <Additional information>.

Explanation: The requested operation is trying to retrieve information about an entity with a key that is invalid. This may occur if the entity has been deleted by another session. The <Message> describes the error that

occurred and the <Throwable> is the type of error that was caught. <Additional information> may provide further information, if available.

User Response: Ask the client to close existing sessions and attempt the operation in a new browser session. If the problem persists, please provide IBM support with a trace of the gui and api components.

UDUC0009E: An invalid value was supplied. Error message: <Message> error: <Throwable>. More information: <Additional information>

Explanation: An invalid value was passed to an API call. The >Message> describes the error that occurred and the <Throwable> is the type of error that was caught. <Additional information> may provide further information, if available.

User Response: Contact IBM support with a trace of the gui and api components.

UDUC0010E: Failed to introspect ActionForm properties. Exception: <Exception>.

Explanation: String properties of a form object could not be introspected which means that the form contents cannot be checked for invalid characters.

User Response: Please contact IBM support with details of the Exception and a trace of the gui component.

UDUC0011E: Failed to invoke reflected methods in ActionForm. Exception: <Exception>.

Explanation: A form object's declared public method for setting or getting a String value could not be invoked. This method is required to check for invalid characters.

User Response: Please contact IBM support with details of the Exception and a trace of the gui component.

UDUC0012E: User console initialization failed to connect to UDDI database. Exception: <Exception>.

Explanation: During user console initialization, connection to the database failed, and threw the exception specified.

User Response: Check the connection to the UDDI database. The included exception message may yield some clues to help you resolve the problem. If unresolved, please contact IBM support with a trace of the gui component during startup.

UDUC0013E: User console initialization failed to initialize tModels. Exception: <Exception>.

Explanation: Indicates that an error has occurred during initialization of ActionServlet, specifically when reading tModels (invoking init method in class TModelNames).

User Response: Check the state of the UDDI database. Visually inspect the TMODEL table and confirm it is populated with valid data. The included exception message may yield some clues to help you resolve the problem. If unresolved, please contact IBM support with a trace of the gui component during startup.

UDUC0014E: User console initialization failed to initialize taxonomies. Exception: <Exception>.

Explanation: Indicates that an error has occurred during initialization of ActionServlet, specifically when reading taxonomy data (invoking init method of CategoryTaxonomyTree).

User Response: Check the state of the UDDI database. The included exception message may yield some clues to help you resolve the problem. If unresolved, please contact IBM support with a trace of the gui component during startup.

UDUT UDDI Utility Tools messages

```
UDUT0001I: Usage: java -jar UDDIUtilityTools.jar '{'function'}' [options]
function:
  -promote entity source   Promote entities between registries
  -export entity source    Extract entities from registry to XML
  -delete entity source    Delete entities from registry
  -import                  Create entities from XML to registry

where entity source is one of:
  -tmodel|-business|-service|-binding key Specify single entity type and key
  -keysFile | -f filename  Specify file containing entity types and keys

options:
  -properties filename     Specify path to configuration file
  -overwrite | -o          Overwrite an entity if it already exists
  -log | -v                Output verbose messages
  -definitionFile filename Specify path to UDDI entity definition file
  -importReferenced        Import entities referenced by source entities

The following options override property settings in configuration file:
  -overwrite
  -log
  -definitionFile
  -importReferenced

Example: java -jar UDDIUtilityTools.jar -promote -keysFile C:/uddikeys.txt
```

Explanation: This is the usage message displayed at the command line when the user has entered an invalid combination of arguments or options.

User Response: Enter the command according to the usage message..

UDUT0002I: ******** Starting UDDI Utility Tools ************

Explanation: This message is used as a marker in the message log file to indicate tool start points.

User Response: None.

UDUT0003I: Promoting entityType<entity type> key<entity key>...

Explanation: Indicates which entity type (business, tModel and so on) is being promoted, and it's key value.

User Response: None.

UDUT0004I: Bad entityType: received<incorrect entity type>, expected <tModel|business|service|binding>

Explanation: The user entered an incorrect entity type.

User Response: Use an entity type of tModel, business, service or binding.

UDUT0005I: Promotion successful.

Explanation: Indicates the promote function completed successfully.

User Response: None.

UDUT0006I: Import successful.

Explanation: Indicates the import function completed successfully.

User Response: None.

UDUT0007I: Export successful.

Explanation: Indicates the export function completed successfully.

User Response: None.

UDUT0008I: Delete successful.

Explanation: Indicates the delete function completed successfully.

User Response: None.

UDUT0009I: Exporting entities ...

Explanation: Indicates the export function has started.

User Response: None.

UDUT0010I: Exporting business, businessKey[<business key].

Explanation: Indicates that the businessEntity with the specified key is being exported.

User Response: None.

UDUT0011I: Exporting service, serviceKey[<service key>].

Explanation: Indicates that the businessService with the specified key is being exported.

User Response: None.

UDUT0012I: Exporting binding, bindingKey[<binding key>].

Explanation: Indicates that the bindingTemplate with the specified key is being exported.

User Response: None.

UDUT0013I: Exporting tModel, tModelKey[<tModel key>].

Explanation: Indicates that the tModel with the specified key is being exported.

User Response: None.

UDUT0014I: Exporting referenced tModel, tModelKey[<tModel key>].

Explanation: Indicates that the referenced tModel with the specified key is being exported.

User Response: None.

UDUT0015I: Exported <entity count> entities.

Explanation: Indicates that the export function completed, and shows the number of entities exported.

User Response: None.

UDUT0016I: Importing entities ...

Explanation: Indicates the import function has started.

User Response: None.

UDUT0017I: Importing business, businessKey[<business key>].

Explanation: Indicates that the businessEntity with the specified key is being imported.

User Response: None.

UDUT0018I: Importing service, serviceKey[<service key>].

Explanation: Indicates that the businessService with the specified key is being imported.

User Response: None.

UDUT0019I: Importing binding, bindingKey[<binding key>]

Explanation: Indicates that the bindingTemplate with the specified key is being imported.

User Response: None.

UDUT0020I: Importing tModel, tModelKey[<tModel key>].

Explanation: Indicates that the tModel with the specified key is being imported..

User Response: None.

UDUT0021I: Importing referenced tModel, tModelKey[<tModel key>].

Explanation: Indicates that the referenced tModel with the specified key is being imported.

User Response: None.

UDUT0022I: Imported <entity count> entities.

Explanation: Indicates that the import function completed, and shows the number of entities imported.

User Response: None.

UDUT0023I: Deleting entities ...

Explanation: Indicates the delete function has started.

User Response: None.

UDUT0024I: Deleting business, businessKey[<business key>].

Explanation: Indicates that the businessEntity with the specified key is being deleted.

User Response: None.

UDUT0025I: Deleting service, serviceKey[<service key>].

Explanation: Indicates that the businessService with the specified key is being deleted.

User Response: None.

UDUT0026I: Deleting binding, bindingKey[<binding key>].

Explanation: Indicates that the bindingTemplate with the specified key is being deleted.

User Response: None.

UDUT0027I: Deleting tModel, tModelKey[<tModel key>].

Explanation: Indicates that the tModel with the specified key is being deleted.

User Response: None.

UDUT0028I: Deleted <entity count> entities.

Explanation: Indicates that the delete function completed, and shows the number of entities deleted.

User Response: None.

UDUT0029I: Serializing ...

Explanation: Indicates that generation of the Entity Definition File has started.

User Response: None.

UDUT0030I: Serialized entities.

Explanation: Indicates that generation of the Entity Definition File completed successfully.

User Response: None.

UDUT0031I: Deserializing ...

Explanation: Indicates that reading of the Entity Definition File and creation of UDDI entities has started.

User Response: None.

UDUT0032I: Deserialized entities.

Explanation: Indicates that reading of the Entity Definition File and creation of UDDI entities completed successfully.

User Response: None.

UDUT0033I: Function ’<function>’ completed successfully.

Explanation: Indicates the requested function completed successfully.

User Response: None.

UDUT0034W: Function '<function>' did not complete successfully. See message log for further information.

Explanation: Indicates the requested function did not complete successfully.

User Response: The message log may yield further information if the verbose option is on. Check the configuration properties file setting are correct. If that does not identify the problem, try running with trace logging enabled. If that does not yield a solution, contact your IBM support center.

UDUT0035W: Parser error: {0}

Explanation: The XML parser reports a warning about the content of the Entity Definition File.

User Response: Based on the context of the warning message, check the validity of the Entity Definition File.

UDUT0036E: Parser error {0}

Explanation: The XML parser reports an error about the content of the Entity Definition File.

User Response: Based on the context of the error message, check the validity of the Entity Definition File.

UDUT0037E: Unrecognized parser feature: <feature>

Explanation: A parser feature set by the UDDI Utility Tools is not recognized by the parser.

User Response: Check you are using the correct type and level of XML parser. If correct, contact your IBM support center.

UDUT0038E: Unsupported parser feature: <feature>

Explanation: A parser feature set by the UDDI Utility Tools is not supported by the parser.

User Response: Check you are using the correct type and level of XML parser. If correct, contact your IBM support center.

UDUT0039E: Unrecognized parser property: <property>, value: <value>

Explanation: A parser property set by the UDDI Utility Tools is not recognized by the parser.

User Response: Check you are using the correct type and level of XML parser. If correct, contact your IBM support center.

UDUT0040E: Unsupported parser property: <property>, value: <value>

Explanation: A parser property set by the UDDI Utility Tools is not supported by the parser.

User Response: Check you are using the correct type and level of XML parser. If correct, contact your IBM support center.

UDUT0041I: <message>

Explanation: This is a placeholder message used during development only.

User Response: None.

UDUT0042E: Unable to find the configuration file: <filepath>

Explanation: UDDI Utility Tools cannot locate the specified configuration file.

User Response: UDDI Utility Tools looks for a default configuration properties with the file name 'UDDIUtilityTools.properties' in the current directory. Check that the configuration file has this name, or that the argument value supplied with the '-properties' option is pointing at a file that exists.

UDUT0043E: An Exception occurred trying to read the configuration file.
Explanation: The configuration file could not be read.

User Response: Check the file path points to a valid file and that the file does not have the 'hidden' attribute set.

UDUT0044W: Configuration file is missing the '<property name>' property.
Explanation: A required property is missing from the configuration file.

User Response: Add the missing property name and value to the configuration file. Check that the property name is not misspelled.

UDUT0045W: Property: '<property name>' has value '<property value>'. It must be either 'true' or 'false'.
Explanation: A value was given to a property other than 'true' or 'false'.

User Response: Set the property value to 'true' or 'false'.

UDUT0046W: Property: '<property name>' has value '<property value>'. It must be an integer value.
Explanation: A value was given to a property other than an integer value.

User Response: Set the property value to an integer value.

UDUT0047E: Unable to find the keyFile file: <keys file path>
Explanation: The keys file could not be located at the specified path.

User Response: Check the file name and path and correct and that the file exists.

UDUT0048E: Unable to read the keyFile file: <keys file path>
Explanation: The keys file could not be read due to an IO error.

User Response: Check the file's hidden attribute is not set.

UDUT0049E: Unable to write to entity definition file: <entity definition file path>
Explanation: During initialization, the Entity Definition File could not be written to.acl.

User Response: Check the file's read only attribute is not set.

UDUT0050E: Unable to find UDDI Entity definition file: <entity definition file path>
Explanation: The Entity Definition File could not be found at the specified file path.

User Response: Check the file path is correct and that the file exists.

UDUT0051E: Unable to read UDDI Entity definition file: <entity definition file path>
Explanation: The Entity Definition File could not be read due to an IO error.

User Response: Check the file's hidden attribute is not set.

UDUT0052E: Unable to close the message file: <file path>
Explanation: The attempt to close the message file failed.

User Response: The disk might be full. If so, clear some space or direct log output to a different disk.

UDUT0053E: Unable to close the trace file: <file path>
Explanation: The attempt to close the trace log file failed.

User Response: The disk might be full. If so, clear some space or direct log output to a different disk.

UDUT0054E: The logger was unable to find the file: <file path>
Explanation: The UDDI Utility Tools logger could not find the specvified file.

User Response: None.

UDUT0055E: ERROR OCCURRED ...
Explanation: General purpose error message used in development only.

User Response: None.

UDUT0056E: Exception:

Explanation: General purpose message prefix used for reporting exceptions.

User Response: None.

UDUT0057W: Only one function may be specified on the command line.

Explanation: Multiple function commands were entered on the command line.

User Response: Specify one function in accordance with the usage message.

UDUT0058W: No function was specified.

Explanation: UDDI Utility Tools was invoked with no function specified.

User Response: Specify one function in accordance with the usage message.

UDUT0059W: The function: <function> was not recognized.

Explanation: The function value did not match any of the allowed functions.

User Response: Specify one function in accordance with the usage message.

UDUT0060W: The argument ’<argument>’ was not recognized.

Explanation: The argument value does not match any of the allowed arguments.

User Response: Specify arguments in accordance with the usage message.

UDUT0061W: There was a missing value for <argument> argument.

Explanation: An expected value for the specified argument was not supplied.

User Response: Specify a value for the argument in accordance with the usage message.

UDUT0062W: Enexpected argument: <argument> (entity key file is already specified).

Explanation: The entity type argument cannot be specified if the keysFile argument is already specified.

User Response: Specify arguments in accordance with the usage message.

UDUT0063W: Unexpected argument: <argument> (entity key is already specified).

Explanation: The keysFile argument cannot be specified if an entity type argument and key value has already been specified.

User Response: Specify arguments in accordance with the usage message.

UDUT0064W: Argument: <argument> cannot be specified more than once.

Explanation: An argument was specified twice in the same command.

User Response: Specify arguments in accordance with the usage message.

UDUT0065E: No entity keys were specified.

Explanation: A keys file or an entity type and key value must be specified for functions using keys.

User Response: Specify arguments in accordance with the usage message.

UDUT0066E: Could not load Database driver: dbDriver<database driver>.

Explanation: The specified database driver could not be loaded.

User Response: Check the database driver value in the configuration file is valid, and the driver’s class is present in the classpath property.

UDUT0067E: Could not create database connection: dbUrl<database URL. dbUser:<database userid>, (dbPasswd not shown).

Explanation: A connection could not be established with the database at the specified URL with the specified userid.

User Response: Check the database URL, userid and password values are correct n the configuration file, and that the database manager is running.

UDUT0068E: Could not close the database connection.

Explanation: An attempt to close the database connection failed.

User Response: If the problem persists, contact your IBM support center.

UDUT0069E: Could not create minimal entity to tModel.

Explanation: The minimal data necessary for a valid tModel could not be inserted in the target UDDI registry database.

User Response: Check the database URL, userid and password values are correct in the configuration file, and that the database manager is running.

UDUT0070E: Could not create minimal entity for Service.

Explanation: The minimal data necessary for a valid businessService could not be inserted in the target UDDI registry database.

User Response: Check the database URL, userid and password values are correct in the configuration file, and that the database manager is running.

UDUT0071E: Could not create minimal entity for Business.

Explanation: The minimal data necessary for a valid businessEntity could not be inserted in the target UDDI registry database.

User Response: Check the database URL, userid and password values are correct in the configuration file, and that the database manager is running.

UDUT0072E: Could not create minimal entity for Binding.

Explanation: The minimal data necessary for a valid bindingTemplate could not be inserted in the target UDDI registry database.

User Response: Check the database URL, userid and password values are correct in the configuration file, and that the database manager is running.

UDUT0073E: There was an error while trying to create an XML Document.

Explanation: An attempt to create the Entity Definition File failed.

User Response: Check the file path specified in the configuration file for the Entity Definition File is valid and is not set to read only.

UDUT0074E: There was an error parsing the entity definition file.

Explanation: An unspecified error occurred when parsing the Entity Definition File.

User Response: Check the entity definition file content is valid according to the UDDI Utility Tools schema file, promoter.xsd.

UDUT0075E: One or more errors occurred while parsing the entity definition file. See message log for details.

Explanation: Errors occurred when parsing the Entity Definition File.

User Response: Check the entity definition file content is valid according to the UDDI Utility Tools schema file, promoter.xsd.

UDUT0076W: One or more warnings were raised while parsing the entity definition file. See message log for details.

Explanation: Warnings occurred when parsing the Entity Definition File.

User Response: Check the entity definition file content is valid according to the UDDI Utility Tools schema file, promoter.xsd.

UDUT0078E: Unable to obtain authinfo.

Explanation: EEE.

User Response: None.

UDUT0079E: The inquiryURL is malformed: <inquiry URL>.

Explanation: The inquiry URL specified in the configuration file is not valid.

User Response: Correct the value for the inquiry URLs (fromInquityURL and toInquiryURL) in the configuration file.

UDUT0080E: The publisherURL is malformed: <publish URL>

Explanation: The publish URL specified in the configuration file is not valid.

User Response: Correct the value for the publish URL (toPublishURL) in the configuration file.

UDUT0081E: Could not get tModel detail for tModelKey[<tModel key>].

Explanation: The get tModel operation failed on the source registry.

User Response: Check the key exists in the source registry.

UDUT0082E: Could not get service detail for serviceKey[<service key>].

Explanation: The get service operation failed on the source registry.

User Response: Check the key exists in the source registry.

UDUT0083E: Could neet get business detail for businessKey[<business key>].

Explanation: The get business operation failed on the source registry.

User Response: Check the key exists in the source registry.

UDUT0084E: Could not get binding detail for bindingKey[<binding key>].

Explanation: The get binding operation failed on the source registry.

User Response: check the key exists in the source registry.

UDUT0085E: Could not save tModel for tModelKey[<tModel key>].

Explanation: The publish operation failed at the target registry.

User Response: Check the tModel is not referencing another entity (such as a tModel) that is not present in the target registry. This may occur if the 'importReferenced' property is set to false. Specify referenced tModels in the referencedtModels section of the Entity Definition File and set 'importReferenced' property in the configuration file to 'true'.

UDUT0086E: Could not save business for businessKey[<business key>].

Explanation: The publish operation failed at the target registry.

User Response: Check the businessEntity is not referencing another entity (such as a tModel) that is not present in the target registry. This may occur if the 'importReferenced' property is set to false. Specify referenced tModels in the referencedtModels section of the Entity Definition File and set 'importReferenced' property in the configuration file to 'true'.

UDUT0087E: Could not save service for parent businessKey[<business key>].

Explanation: The publish operation failed at the target registry.

User Response: Check the businessEntity specified as the parent of the businessService exists in the target registry.

UDUT0088E: Could not save binding for parent serviceKey[<service key>].

Explanation: The publish operation failed at the target registry.

User Response: Check the businessService specified as the parent of the bindingTemplate exists in the target directory.

UDUT0089W: Did not save service for serviceKey[<service key>].

Explanation: The parent business for the specified businessService does not exist.

User Response: Check the key value for the parent entity is correct in the Entity Definition File, and that the entity exists in the target registry.

UDUT0090W: Did not save binding for bindingKey[<binding key>].

Explanation: The parent service for the specified bindingTemplate does not exist.

User Response: Check the key value for the parent entity is correct in the Entity Definition File, and that the entity exists in the target registry.

UDUT0091E: Could not delete business for businessKey[<businness key>].

Explanation: The UDDI4J operation to delete the businessEntity with the specified key failed.

User Response: Check the userid and password property values in the configuration file and that the entity exists in the target UDDI registry.

UDUT00921E: Could not delete service for serviceKey[<tModel key>].

Explanation: The UDDI4J operation to delete the businessService with the specified key failed.

User Response: Check the userid and password property values in the configuration file and that the entity exists in the target UDDI registry.

UDUT0093E: Could not delete binding for bindingKey[<binding key>].

Explanation: The UDDI4J operation to delete the bindingTemplate with the specified key failed.

User Response: Check the userid and password property values in the configuration file and that the entity exists in the target UDDI registry.

UDUT0094E: Could not delete tModel for tModelKey[<tModel key>].

Explanation: The UDDI4J operation to delete the tModel with the specified key failed.

User Response: Check the userid and password property values in the configuration file and that the entity exists in the target UDDI registry.

UDUT0096W: <entity type><key value> is not a valid UUID.

Explanation: The key value entered does not comply with the format specified for a UUID in the UDDI specification.

User Response: Enter a valid UUID key.

UDUT0097W: Did not save tModel for tModelKey[<tModel key>] as it already exists. Use the -overwrite argument to overwrite the tModel.

Explanation: The tModel was not saved because the overwrite property is false

User Response: If the desired action is to overwrite existing entities, specify -overwrite on the command line or set the overwrite property in the configuration file to true.

UDUT0098W: Did not save business for businessKey[<business key>] as it already exists. Use the -overwrite argument to overwrite the tModel.

Explanation: The businessEntity was not saved because the overwrite property is false

User Response: If the desired action is to overwrite existing entities, specify -overwrite on the command line or set the overwrite property in the configuration file to true.

UDUT0099W: Did not save service for serviceKey[<service key key>] as it already exists. Use the -overwrite argument to overwrite the tModel.

Explanation: The businessService was not saved because the overwrite property is false

User Response: If the desired action is to overwrite existing entities, specify -overwrite on the command line or set the overwrite property in the configuration file to true.

UDUT0100W: Did not save binding for bindingKey[<binding key key>] as it already exists. Use the -overwrite argument to overwrite the tModel.

Explanation: The bindingTemplate was not saved because the overwrite property is false

User Response: If the desired action is to overwrite existing entities, specify -overwrite on the command line or set the overwrite property in the configuration file to true.

UDUT0101W: Bad entity type: received<entity type>, expected<tModel|business|service|binding>.

Explanation: The entered entity type was not recognized.

User Response: Specify arguments in accordance with the usage message.

UDUT0102E: Promotion failed.

Explanation: The promote function failed to complete.

User Response: Check the configuration properties file has correct settings.

UDUT0106E: Unable to commit transaction.

Explanation: The insertion of minimal entity data during the import function failed to commit to the database.

User Response: Check the database configuration. If necessary, turn on trace logging and look for the SQLException that is recorded.

UDUT0107E: Unable to set auto-commit off on the database connection.

Explanation: UDDI Utility Tools needs to control commits of data changes, however the attempt to turn off auto-commit failed.

User Response: Check the database configuration. If necessary, turn on trace logging and look for the SQLExecption that is recorded.

UDUT0109E: The import function requires a UDDI entity definition file to be specified.

Explanation: A required argument value was not specified.

User Response: Specify -definition <path to Entity Definition File> on the command line, or set the value of the UDDIEntityDefinitionFile property in the configuration file to the path to the Entity Definition File.

UDUT0110E: A cyclic dependency exists in the referenced tModels. The reference from tModel with key [<tModel key>] to the tModel with key [<tModel key>] completes the detected cycle.

Explanation: A cycle has been detected such that a tModel is being referenced by a tModel that it directly or indirectly references. This would cause the UDDI Utility Tools to enter an infinite loop trying to import referenced tModels, so the process is halted.

User Response: Edit the Entity Definition File and temporarily remove the reference to the tModel in the cycle, taking a note of the referenced details. After the import has successfully completed, update the tModel in the target registry to reintroduce the reference you previously removed. this can be done using the UDDI User Console, UDDI4J, or by creating a new Entity Definition File with just the tModel to be updated, and running the UDDI Utility Tools with the import function.

UDUT0112E: An unexpected exception has occurred: <Exception message>.

Explanation: An unexpected error occurred.

User Response: Check configuration file settings and all registries and databases are active. If necessary, contact your IBM support center.

UDUT0113E: Could not get a response from UDDI registry at URL: <URL>.

Explanation: A TransPortException occurred while performing an UDDI4J operation on the UDDI registry at the specified URL.

User Response: Check configuration properties for the UDDI registry in question and ensure the UDDI registry is active.

UDUT0114E: An IOException occurred trying to invoke 'java'.

Explanation: When UDDI Utility Tools was invoked using the java -jar syntax, the invocation of the second JVM failed.

User Response: Check configuration property 'classpath' value is correct, and that Java is configured to run from the command line.

UDUT0115I: Imported <entity count> entities and <referenced entity count> referenced entities.

Explanation: Indicate that the import step of the import or promote function has completed, showing the number of entities imported.

User Response: None.

UDUT0116W: Not all minimal entities could be removed. The following remain in the database:

Explanation: A publish step was not successful which may have left one or more minimal entities in the target registry database. UDDI Utility Tools attempts to remove these minimal entities but in this case, the removal has failed. Following messages will indicate which minimal entities are left in the target registry.

User Response: You can attempt to remove the minimal entites using normal methods, such as the user console, UDDI4J, or using the delete function of the UDDI Utility Tools.

UDUT0117W: Business minimal entities with businessKey [<business key>] has not been removed from the database.

Explanation: A business minimal entity was orphaned in the target registry database and attempts to remove it failed.

User Response: Identify the orphaned minimal entity in the target and attempt to remove using normal UDDI methods, or by using the delete function of the UDDI Utility Tools.

UDUT0118W: Service minimal entity with serviceKey [<service key>] has not been removed from the database.

Explanation: A service minimal entity was orphaned in the target registry database and attempts to remove it failed.

User Response: Identify the orphaned minimal entity in the target registry and attempt to remove using normal UDDI delete methods, or by using the delete function of the UDDI Utility Tools.

UDUT0119W: Binding Template minimal entity with bindingKey [<binding key>] has not been removed from the database.

Explanation: A binding minimal entity was orphaned in the target registry database and attempts to remove it failed.

User Response: Identify the orphaned minimal entity in the target registry and attempt to remove using normal UDDI delete methods, or by using the delete function of the UDDI Utility Tools.

UDUT0120W: TModel minimal entity with tModelKey [<tModel key>] has not been removed from the database.

Explanation: A tModel minimal entity was orphaned in the target registry database and attempts to remove it failed.

User Response: Identify the orphaned minimal entity in the target registry and attempt to remove using normal UDDI delete methods, or by using the delete function of the UDDI Utility Tools.

UDUT0121I: Created business minimal entity with businessKey [<business key>].
Explanation: Indicates the minimal data required for a businessEntity has successfully been inserted in the target UDDI registry database.

User Response: None.

UDUT0122I: Created service minimal entity with serviceKey [<service key>].
Explanation: Indicates the minimal data required for a businessService has successfully been inserted in the target UDDI registry database.

User Response: None.

UDUT0123I: Created binding template minimal entity with bindingKey [<binding key>].
Explanation: Indicates the minimal data required for a bindingTemplate has successfully been inserted in the target UDDI registry database.

User Response: None.

UDUT0124I: Created tModel minimal entity with tModelKey [<tModel key>].
Explanation: Indicates the minimal data required for a tModel has successfully been inserted in the target UDDI registry database.

User Response: None.

UDUT0125I: Deleted business minimal entity with businessKey [<business key>].
Explanation: Indicates the minimal data inserted for a businessEntity was successfully removed from the target UDDI registry database. This would normally happen after a publish operation has failed.

User Response: None.

UDUT0126I: Deleted service minimal entity with serviceKey [<service key>].
Explanation: Indicates the minimal data required for a businessService was successfully removed from the target UDDI registry database. This would normally happen after a publish operation has failed.

User Response: None.

UDUT0127I: Deleted binding template minimal entity with bindingKey [<binding key>].
Explanation: Indicates the minimal data required for a bindingTemplate was successfully removed from the target UDDI registry database. This would normally happen after a publish operation has failed.

User Response: None.

UDUT0128I: Deleted tModel minimal entity with tModelKey [<tModel key>].
Explanation: Indicates the minimal data required for a tModel was successfully removed from the target UDDI registry database. This would normally happen after a publish operation has failed.

User Response: None.

UDUT0129E: Find related businesses failed.
Explanation: The UDDI4J find related businesses operation did not complete.

User Response: Check the configuration properties for the source registry, such as fromInquiryURL.

UDUT0130E: Find businesses failed.
Explanation: The UDDI4J find businesses operation did not complete.

User Response: Check the configuration properties for the source registry, such as fromInquiryURL.

UDUT0131E: Find services failed.
Explanation: The UDDI4J find services operation did not complete.

User Response: Check the configuration properties for the source registry, such as fromInquiryURL.

UDUT0132E: Find tModels failed.

Explanation: The UDDI4J find tModels operation did not complete.

User Response: Check the configuration properties for the source registry, such as fromInquiryURL.

UDUT0133E: Find bindings failed.

Explanation: The UDDI4J find bindings operation did not complete.

User Response: Check the configuration properties for the source registry, such as fromInquiryURL.

UDUT0134I: Performing inquiry request ...

Explanation: Indicated the find operation for selecting keys has started.

User Response: None.

UDUT0135I: Extracted keys from inquiry results.

Explanation: Indicates the find operation to select keys has completed successfully.

User Response: None

UDUU (Web Services UDDI) messages

There are no messages issued by this component.

Running the UDDI samples

The UDDI samples, and documentation on how to use them, are available through the Web Services UDDI samples link on the Samples Central page of the IBM WebSphere Developer Domain Web site.

Installation Verification Program (IVP)

There are some samples available on the WebSphere Developer Domain (WSDD) web site (at http://www7b.software.ibm.com/wsdd/library/samples/AppServer.html) that are intended to provide an optional Installation Verification test, or IVP, for the UDDI Registry component.

This topic describes how to run these installation verification programs (IVPs) to verify that the IBM UDDI Registry has been installed correctly.

There are two IVP SOAP samples: SOAPSampleIVPa and SOAPSampleIVPb. They are intended to verify the successful installation of the product, and should be used in conjunction with the UDDI Users Console (GUI). SOAPSampleIVPa saves some data to the registry which you can then find using the GUI. Finally you can delete the data by running SOAPSampleIVPb.

The IVP samples are installed into the same target directory as the other SOAP samples and they use the same XML files as the basic Java SOAP samples.

SOAPSampleIVPa saves three businesses, six services (2 per business) and three tModels. The data structures are very basic and consist only of a name. The keys returned by the save_* UDDI API calls are then written to a file, `SOAPSampleIVPa.out`. SOAPSampleIVPb then reads in these keys from the file to delete the saved data from the UDDI registry.

Note: Each time you run SOAPSampleIVPa, it overwrites the output file `SOAPSampleIVPa.out` so, if you wish to use SOAPSampleIVPb to delete the data, you must run this before you next run SOAPSampleIVPa.

Note: As supplied, the IVP programs are written to work on a system without authentication. It is possible to configure the IVPs to work with authentication (see http://www7b.software.ibm.com/wsdd/library/samples/AppServer.html), however, if possible it is recommended you run them on a non-authenticated system.

Steps for this task

Perform the following steps on the same system as the UDDI Registry:

1. Ensure that DB2 and the WebSphere Administrative Server are started.
2. Start the WebSphere Administrator's Console and ensure the default server is started and the UDDI Registry Application is started.
3. For SOAP samples to work, you need to ensure that the Client Developer Kit for Java is either the one shipped with IBM WebSphere Application Server or a later IBM Developer Kit for Java.:
   - For Windows - ensure that <WebSphere-install-dir>\java\bin is present in the PATH statement before any other Developer Kits for Java
   - For Unix Platforms - ensure that <WebSphere-install-dir>/java/bin is present in the PATH statement before any other Developer Kits for Java

   Note: You **must** use the IBM WebSphere supplied Developer Kit for Java or a later level of the IBM Developer Kit for Java.

   For Windows, the default system path can be set via **Control Panel ...-> Settings ...-> System ...-> Advanced Properties ...-> Environment Variables**

   Alternatively, this can be accomplished just for the shell where you plan to run the samples by modifying the path within the shell:
   - For Windows - set path=<WebSphere-install-dir>\java\bin;%path%
   - For Unix Platforms - export PATH=<WebSphere-install-dir>/java/bin:$PATH
4. Copy the samples and *.xml files to a directory
5. Compile both SOAPSampleIVPa and SOAPSampleIVPb by entering (from a command prompt):

   'javac SOAPSampleIVPa.java'

   and

   'javac SOAPSampleIVPb'
6. Run SOAPSampleIVPa by entering 'java SOAPSampleIVPa'. This should publish a number of businesses and services and technical models into the registry.
7. Start your Web browser on the same system as the UDDI Registry.
8. To display the UDDI GUI home page, enter the following URL:
   - `http://localhost:9080/uddigui`
9. On the find page, complete the following steps:
   a. Select the business radio button
   b. In the data entry field, enter `%` (the wild card symbol)
   c. Click **Find**

You should get a results page returned with three businesses (mybusiness1, mybusiness2, and mybusiness3). This demonstrates that the API and the UDDI user console are working correctly.

10. To see the services that are available for a business, click the ″Show Services″ option next to the business.
11. To delete all of the IVP data, run SOAPSampleIVPb (from the command prompt as before - by entering ′java SOAPSampleIVPb′)
12. On the find page, complete the following steps:
    a. Select the business radio button
    b. In the data entry field, enter % (the wild card symbol)
    c. Click **Find**

    You should get an empty results page returned.

Reporting problems with the IBM WebSphere UDDI Registry

If you report a problem with the IBM WebSphere UDDI Registry component to IBM, supply the following information:

1. A detailed description of the problem.
2. The build date and time of the version you are using. This can be obtained as follows:
   - In the installedApps subdirectory of the WebSphere installation location, you will find a subdirectory called UDDI_Registry.<nodename>.<servername>.ear, where <nodename> is the name of the node into which the UDDI Registry application is installed, and <servername> is the name of the server. Within that subdirectory, you will find a file called version.txt. Include the contents of this file as part of your information.
   - If the UDDI Registry has been started with tracing enabled for the UDDI component, you should find a trace entry in the WebSphere trace log that includes the strings ″getUDDIMessageLogger″ and ″UDDI Build :″ followed by the build date and time, and the build system. Also include this information.
3. Any relevant log files and trace files.
   - If the problem occurred while setting up and installing the UDDI Registry application using one of the setup scripts, setupuddi.jacl or appserversetupuddi.jacl, supply the log output from running the script. (If you did not redirect the output from the script file to a log file, rerun the script, this time redirecting the output as described in the section ′Installing and Setting up a UDDI Registry′.) The log file is written to the directory from which you ran the setup script.
   - If the problem occurred while removing the UDDI Registry application using one of the remove scripts, removeuddi.jacl or appserverremoveuddi.jacl, supply the log output from running the script. (If you did not redirect the output from the script file to a log file, rerun the script, this time redirecting the output as described in the section ′Removing the UDDI Registry from a deployment manager cell′ or ′Removing the UDDI Registry application from a single appserver′.) The log file is written to the directory from which you ran the remove script.
   - If the problem occurred while creating the UDDI Registry database using the UDDI DB2 Setup Wizard, supply the log file UDDIloadDB.log, which is written to the directory from which the wizard was run.
   - If the problem occurred while running the UDDI Registry, enable UDDI tracing (if not already enabled) and supply the trace log from the logs directory of the application server on which the UDDI Registry was running. See ′Turning on UDDI Trace′ for details on how to enable UDDI tracing.

- Also supply the WebSphere log files system.out and system.err.
- Supply details of the version of IBM WebSphere Application Server you are running by executing the command `versioninfo (Windows) or versioninfo.sh (Unix platforms)` on both the application server and deployment manager nodes and directing the output to a log file.

4. If appropriate, any application code that you are using and the output produced by the application code.

In addition to the above, it is useful to run the WebSphere collector tool and send the resulting jar file(s) (two files if run from base application server AND DeploymentManager) to IBM. See Running the collector tool

Feedback

See the section on ″Obtaining help from IBM″ elsewhere in this InfoCenter for details on seeking assistance.

Chapter 10. Class loading

Class loaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers.

1. Read about class loaders. The article ""Class loading: Resources for learning" on page 723" points to additional sources.
2. If necessary, migrate class-loader Module Visibility Mode settings for Version 4.0.x applications to Version 5.0 application or WAR class-loader policies.
3. **5.0.1** **5.0.2** When assembling an enterprise application resource (EAR) file that has EJB modules, set the class path for the class loader to use during packaging.
4. If an application module uses a resource, create a resource provider that specifies the directory name of the resource drivers. Do not specify the resource JAR file names. All JAR files in the specified directory will be added into the class path of the WebSphere Application Server extensions class loader.
5. Configure class loaders of an application server for the run-time environment.
   a. Click **Servers > Application Servers >***server_name* and, on the settings page for an application server, set the application class-loader policy and application class-loader mode.

      The application class-loader policy controls the isolation of applications running in the system. When set to SINGLE, applications are not isolated; a single application class loader is used to contain all EJB modules, dependency JAR files, and shared libraries in the system. When set to MULTIPLE, applications are isolated from each other; each application receives its own class loader to load that application's EJB modules, dependency JAR files, and shared libraries.

      The application class-loader mode specifies the class-loader mode when the application class-loader policy is SINGLE. PARENT_FIRST causes the class loader to first delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. PARENT_LAST causes the class loader to first attempt to load classes from its local class path before delegating the class loading to its parent. This allows an application class loader to override and provide its own version of a class that exists in the parent class loader.
   b. On the settings page for an application server, click **Classloader**. On the Classloader page, click **New**.
   c. On the settings page for a class loader, specify the class-loader mode. PARENT_FIRST causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local classpath. PARENT_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent. Then, click **OK**.
   d. On the settings page for a class loader, click **Libraries**. From the Library Ref page, click **Add**. On the settings page for a library reference, specify variables for the library reference as needed and click **OK**. Repeat the previous step until you define a library reference instance for each library file that your application needs. To define a library reference, you must first define one or more shared libraries.
6. When configuring an installed Web module for deployment in the run-time environment, set the class-loader mode.

© Copyright IBM Corp. 2003

Class loaders

Class loaders are part of the Java virtual machine (JVM) code and are responsible for finding and loading class files. Class loaders affect the packaging of applications and the run-time behavior of packaged applications deployed on application servers.

The run-time environment of WebSphere Application Server uses the following class loaders to find and load new classes for an application in the following order:

1. The **bootstrap, extensions, and CLASSPATH class loaders** created by the JVM.

   The bootstrap class loader uses the boot classpath (typically classes in jre/lib) to find and load classes. The extensions class loader uses the system property java.ext.dirs (typically jre/lib/ext) to find and load classes. The CLASSPATH class loader uses the CLASSPATH environment variable to find and load classes.

   The CLASSPATH class loader contains the J2EE APIs of the WebSphere Application Server product (inside j2ee.jar). Because the J2EE APIs are in this class loader, you can add libraries that depend on J2EE APIs to the classpath system property to extend a server's classpath. However, a preferred method of extending a server's classpath is to add a shared library.
2. A **WebSphere-specific extensions class loader**.

   The WebSphere extensions class loader loads the WebSphere run-time and J2EE classes that are required at run time. The extensions class loader uses a ws.ext.dirs system property to determine the path used to load classes. Each directory in the ws.ext.dirs classpath and every JAR file or ZIP file in these directories is added to the classpath used by this class loader.

   The WebSphere extensions class loader also loads resource provider classes into a server if an application module installed on the server refers to a resource that is associated with the provider and if the provider specifies the directory name of the resource drivers.
3. One or more **application module class loaders** that load elements of enterprise applications running in the server.

   The application elements can be Web modules, EJB modules, resource adapters, and dependency JAR files. Application class loaders follow J2EE class-loading rules to load classes and JAR files from an enterprise application. The WebSphere run time enables you to associate a shared library classpath with an application.

Each class loader is a child of the class loader above it. That is, the application module class loaders are children of the WebSphere-specific extensions class loader, which is a child of the CLASSPATH Java class loader. Whenever a class needs to be loaded, the class loader usually delegates the request to its parent class loader. If none of the parent class loaders can find the class, the original class loader attempts to load the class. Requests can only go to a parent class loader; they cannot go to a child class loader. If the WebSphere class loader is requested to find a class in a J2EE module, it cannot go to the application module class loader to find that class and a ClassNotFoundException occurs. Once a class is loaded by a class loader, any new classes that it tries to load reuse the same class loader or go up the precedence list until the class is found.

Class-loader isolation policies

The number and function of the application module class loaders depends on the class-loader policies specified in the server configuration. Class loaders provide

multiple options for isolating applications and modules to enable different application packaging schemes to run on an application server.

Two class-loader policies control the isolation of applications and modules:

Application class-loader policy
: Application class loaders consist of EJB modules, dependency JAR files, resource adapters, and shared libraries. Depending on the application class-loader policy, an application class loader can be shared by multiple applications (SINGLE) or unique for each application (MULTIPLE). The application class-loader policy controls the isolation of applications running in the system. When set to SINGLE, applications are not isolated. When set to MULTIPLE, applications are isolated from each other.

WAR class-loader policy
: By default, Web module class loaders load the contents of the WEB-INF/classes and WEB-INF/lib directories. The application class loader is the parent of the Web module class loader. You can change the default behavior by changing the application's WAR class-loader policy.

  The WAR class-loader policy controls the isolation of Web modules. If this policy is set to APPLICATION, then the Web module contents also are loaded by the application class loader (in addition to the EJB files, RAR files, dependency JAR files, and shared libraries). If the policy is set to MODULE, then each web module receives its own class loader whose parent is the application class loader.

Note: WebSphere server class loaders never load application client modules.

For each application server in the system, you can set the application class-loader policy to SINGLE or MULTIPLE. When the application class-loader policy is set to SINGLE, then a single application class loader loads all EJB modules, dependency JAR files, and shared libraries in the system. When the application class-loader policy is set to MULTIPLE, then each application receives its own class loader used for loading that application's EJB modules, dependency JAR files, and shared libraries.

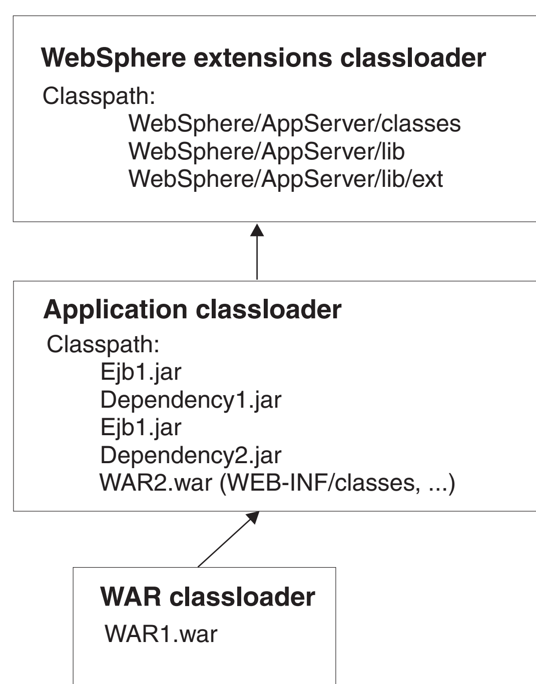
This application class loader can load each application's Web modules if that WAR module's class-loader policy is also set to APPLICATION. If the WAR module's class-loader policy is set to APPLICATION, then the application's loader loads the WAR module's classes. If the WAR class-loader policy is set to MODULE, then each WAR module receives its own class loader.

The following example shows that when the application class-loader policy is set to SINGLE, a single application class loader loads all EJB modules, dependency JAR files, and shared libraries of all applications on the server. The single application class loader can also load Web modules if an application has its WAR class-loader policy set to APPLICATION. Applications having a WAR class-loader policy set to MODULE use a separate class loader for Web modules.

```
Application class-loader policy: SINGLE

Application 1
 Module:  EJB1.jar
 Module: WAR1.war
   MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = MODULE
Application 2
```

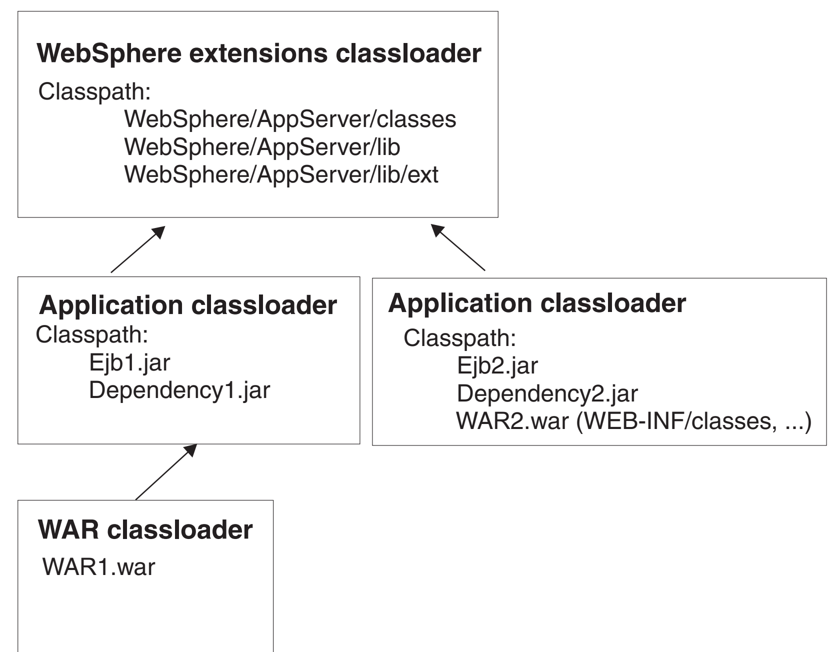
```
Module:   EJB2.jar
 MANIFEST Class-Path: Dependency2.jar
Module: WAR2.war
 WAR Classloader Policy = APPLICATION
```



The following example shows that when the application class-loader policy of an application server is set to MULTIPLE, each application on the server has its own class loader. An application class loader also loads its Web modules if the application's WAR class-loader policy is set to APPLICATION. If the policy is set to MODULE, then a Web module uses its own class loader.

```
Application class-loader policy: MULTIPLE

Application 1
 Module:  EJB1.jar
 Module: WAR1.war
  MANIFEST Class-Path: Dependency1.jar
  WAR Classloader Policy = MODULE
Application 2
 Module:   EJB2.jar
  MANIFEST Class-Path: Dependency2.jar
 Module: WAR2.war
  WAR Classloader Policy = APPLICATION
```



Class-loader modes

There are two possible values for a class-loader mode:

PARENT_FIRST
: The PARENT_FIRST class-loader mode causes the class loader to first delegate the loading of classes to its parent class loader before attempting to load the class from its local classpath. This is the default for class-loader policy and for standard JVM class loaders.

PARENT_LAST
: The PARENT_LAST class-loader mode causes the class loader to first attempt to load classes from its local classpath before delegating the class loading to its parent. This policy allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

The following settings determine a class loader's mode:

- If the application class-loader policy of an application server is SINGLE, the application class-loader policy of an application server defines the mode for an application class loader.
- If the application class-loader policy of an application server is MULTIPLE, the class-loader mode of an application defines the mode for an application class loader.
- If the WAR class-loader policy of an application is MODULE, the WAR class-loader policy of a Web module defines the mode for a WAR class loader.

Class loader collection

Use this page to manage class-loader instances on an application server. A class loader determines whether an application class loader or a parent class loader finds and loads Java class files for an application.

To view this administrative console page, click **Servers > Application Servers >***server_name***> Classloader**.

Classloader ID

States a string unique to the server identifying the class-loader instance. The product assigns the identifier.

Classloader Mode

Specifies the class-loader mode when the application class-loader policy is SINGLE. PARENT_FIRST causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. PARENT_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent; this allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

Class loader settings

Use this page to configure a class loader for applications that reside on an application server.

To view this administrative console page, click **Servers > Application Servers >***server_name* **> Classloader >***class_loader_ID*.

Classloader ID

States a string unique to the server identifying the class-loader instance. The product assigns the identifier.

Data type	String

Classloader Mode

Specifies the class-loader mode when the application class-loader policy is SINGLE. PARENT_FIRST causes the class loader to delegate the loading of classes to its parent class loader before attempting to load the class from its local class path. PARENT_LAST causes the class loader to attempt to load classes from its local class path before delegating the class loading to its parent; this allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

Data type	String
Default	PARENT_FIRST

Migrating the class-loader Module Visibility Mode setting

WebSphere Application Server Version 4.0.x had a server-wide configuration setting called **Module Visibility Mode**. For Version 5.0, you use application or WAR class-loader policies instead of module visibility modes. The Version 5.0 policies provide additional flexibility because you can configure applications running in a server for an application class-loader policy of SINGLE or MULTIPLE and for a WAR class-loader policy of APPLICATION or MODULE.

To migrate module visibility modes in your Version 4.0.x applications to their equivalents in Version 5.0, change the settings for your Version 4.0.x applications and modules to the Version 5.0 values shown in the table below.

Version 4.0.x module visibility mode	Version 5.0 application class-loader policy	Version 5.0 WAR class-loader policy
Server	SINGLE	APPLICATION

Compatibility	SINGLE	MODULE
Application	MULTIPLE	APPLICATION
Module*	MULTIPLE	MODULE
J2EE	MULTIPLE	MODULE

*There is no exact equivalent for the Version 4.0.x Module mode because it isolated EJB modules within an application.

Class loading: Resources for learning

Use the following links to find relevant supplemental information about class loaders. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

View links to additional information about:
- Programming model and decisions
- Programming instructions and examples
- Programming specifications

Programming model and decisions
- J2EE Class Loading Demystified
- Understanding J2EE Application Server Class Loading Architectures

Programming instructions and examples
- Developing and Deploying Modular J2EE Applications with WebSphere Studio Application Developer and WebSphere Application Server
- IBM WebSphere Application Server Programming

Programming specifications
- Sun's J2EE™ Platform Specification
- Sun's J2EE™ Extension Mechanism Architecture

Chapter 11. Using EJB query

The EJB query language is used to specify a query over container-managed entity beans. The language is similar to SQL. An EJB query is independent of the bean's mapping to a persistent store.

An EJB query can be used in three situations:
- To define a finder method of an EJB entity bean.
- To define a select method of an EJB entity bean.
- To dynamically specify a query using the `executeQuery()` dynamic API.

Finder and select queries are specified in the bean's deployment descriptor using the `<ejb-ql>` tag. Queries specified in the deployment descriptor are compiled into SQL during deployment. Dynamic queries require the interface provided by WebSphere Application Server Enterprise.

WebSphere's EJB query language is compliant with the EJB QL defined in Sun's EJB 2.0 specification and has additional capabilities as listed in the topic Comparison of EJB 2.0 specification and WebSphere Query Language.

In your WebSphere application, you can define an EJB query in the following ways:
- **Application Assembly Tool.** When assembling an EJB 2.0 entity bean, specify the `<ejb-ql>` tag for the finder() or select() method.
- **WebSphere Studio Application Developer**. When defining an entity bean, specify the `<ejb-ql>` tag for the finder or select method.
- **Dynamic query service.** Add the executeQuery() method to your application. The dynamic query API is provided as an Enterprise Extension to WebSphere Application Server.

Before using EJB query, familiarize yourself with query language concepts, starting with the topic, EJB Query Language.

See the topic Example: EJB queries.

EJB query language

An EJB query is a string that contains the following elements:
- a SELECT clause that specifies the EJBs or values to return;
- a FROM clause that names the bean collections;
- an optional WHERE clause that contains search predicates over the collections;
- an optional GROUP BY and HAVING clause (see Aggregation functions);
- an optional ORDER BY clause that specifies the ordering of the result collection.

The SELECT clause is optional in order to maintain compatibility with WebSphere Application Server Version 4.

Collections of entity beans are identified in EJB queries through the use of their abstract schema name in the query FROM clause.

The elements of EJB query language are discussed in more detail in the following related topics.

© Copyright IBM Corp. 2003

Example: EJB queries

Here is an example EJB schema, followed by a set of example queries:

Table 5. DeptBean schema

Entity bean name (EJB name)	DeptEJB (not used in query)
Abstract schema name	DeptBean
Implementation class	com.acme.hr.deptBean (not used in query)
Persistent attributes (cmp fields)	• deptno - Integer (key) • name - String • budget - BigDecimal
Relationships	• emps - 1:Many with EmpEJB • mgr - Many:1 with EmpEJB

Table 6. EmpBean schema

Entity bean name (EJB name)	EmpEJB (not used in query)
Abstract schema name	EmpBean
Implementation class	com.acme.hr.empBean (not used in query)
Persistent attributes (cmp fields)	• empid - Integer (key) • name - String • salary - BigDecimal • bonus - BigDecimal • hireDate - java.sql.Date • birthDate - java.util.Calendar • address - com.acme.hr.Address
Relationships	• dept - Many:1 with DeptEJB • manages - 1:Many with DeptEJB

Address is a serializable object used as cmp field in EmpBean. The definition of address is as follows:

```
   public class com.acme.hr.Address  extends Object implements Serializable {
public String street;
public String state;
public String city;
public Integer zip;
     public double distance (String start_location) { ... } ;
     public  String format ( ) { ... } ;
}
```

The following query returns all departments:

```
SELECT OBJECT(d) FROM DeptBean d
```

The following query returns departments whose name begins with the letters "Web". Sort the result by name:

```
SELECT OBJECT(d) FROM DeptBean d WHERE  d.name LIKE  'Web%' ORDER BY d.name
```

The keywords SELECT and FROM are shown in uppercase in the examples but are case insensitive. If a name used in a query is a reserved word, the name must be enclosed in double quotes to be used in the query. There is a list of reserved words later in this document. Identifiers enclosed in double quotes are case sensitive. This example shows how to use a cmp field that is a reserved word:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d."select" > 5
```

The following query returns all employees who are managed by Bob. This example shows how to navigate relationships using a path expression:

```
SELECT OBJECT (e) FROM EmpBean e WHERE e.dept.mgr.name='Bob'
```

A query can contain a parameter which referes to the corresponding value of the finder or select method. Query parameters are numbered starting with 1:

```
SELECT OBJECT (e) FROM EmpBean e WHERE e.dept.mgr.name= ?1
```

This query shows navigation of a multivalued relationship and returns all departments that have an employee that earns at least 50000 but not more than 90000:

```
SELECT OBJECT(d) FROM DeptBean d,  IN (d.emps) AS e
WHERE e.salary BETWEEN 50000 and 90000
```

There is a join operation implied in this query between each department object and its related collection of employees. If a department has no employees, the department does not appear in the result. If a department has more than one employee that earns more than 50000, that department appears multiple times in the result.

The following query eliminates the duplicate departments:

```
SELECT DISTINCT OBJECT(d) from DeptBean d,  IN (d.emps) AS e  WHERE e.salary > 50000
```

Find employees whose bonus is more than 40% of their salary:

```
SELECT OBJECT(e) FROM EmpBean e where e.bonus > 0.40 * e.salary
```

Find departments where the sum of salary and bonus of employees in the department exceeds the department budget:

```
SELECT OBJECT(d) FROM DeptBean d where d.budget <
( SELECT SUM(e.salary+e.bonus) FROM IN(d.emps) AS e )
```

A query can contain DB2 style date-time arithmetic expressions if you use java.sql.* datatypes as CMP fields and your datastore is DB2. Find all employees who have worked at least 20 years as of January 1st, 2000:

```
SELECT OBJECT(e) FROM EmpBean e where year(  '2000-01-01' - e.hireDate ) >= 20
```

If the datastore is not DB2 or if you prefer to use java.util.Calendar as the CMP field, then you can use the java millsecond value in queries. The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Find departments with no employees:

```
SELECT OBJECT(d) from DeptBean d where d.emps IS EMPTY
```

Find all employees whose earn more than Bob:

```
SELECT OBJECT(e) FROM EmpBean e, EmpBean b
WHERE b.name = 'Bob' AND e.salary + e.bonus > b.salary + b.bonus
```

Find the employee with the largest bonus:

```
SELECT OBJECT(e) from EmpBean e  WHERE e.bonus =
(SELECT MAX(e1.bonus) from EmpBean e1)
```

The above queries all return EJB objects. A finder method query must always return an EJB Object for the home. A select method query can in addition return CMP fields or other EJB Objects not belonging to the home.

The following would be valid select method queries for EmpBean. Return the manager for each department:

```
SELECT  d.mgr FROM DeptBean d
```

Return department 42 manager's name:

```
SELECT  d.mgr.name FROM DeptBean d WHERE  d.deptno = 42
```

Return the names of employees in department 42:

```
SELECT e.name FROM EmpBean e WHERE  e.dept.deptno=42
```

Another way to write the same query is:

```
SELECT e.name from DeptBean d, IN (d.emps) AS e WHERE d.deptno=42
```

Finder and select queries allow only a single CMP field or EJBObject in the SELECT clause.

FROM clause

The FROM clause specifies the collections of objects to which the query is to be applied. Each collection is identified either by an abstract schema name and an identification variable, called a range variable, or by a collection member declaration that identifies a multivalued relationship and an identification variable.

Conceptually, the semantics of the query is to first form a temporary collection of tuples R. Tuples are composed of elements from the collections identified in the FROM clause. Each tuple contains one element from each of the collections in the FROM clause. All possible combinations are formed subject to the constraints imposed by the collection member declarations. If any schema name identifies a collection for which there are no records in the persistent store, then the temporary collection R will be empty.

Example: FROM clause

`DeptBean` contains records 10, 20 and 30 in the persistent store. `EmpBean` contains records 1, 2 and 3 that are related to department 10 and records 4, 5 that are related to department 20. Department 30 has no related employees.

```
FROM DeptBean d, EmpBean e
```

This forms a temporary collection R that contains 15 tuples.

```
FROM  DeptBean d,  DeptBean d1
```

This forms a temporary collection R that contains 9 tuples.

```
FROM  DeptBean d,  IN (d.emps) AS e
```

This forms a temporary collection R that contains 5 tuples. Department 30 because it contains no employees will not be in R. Department 10 will be contained in R three times and department 20 will be contained in R twice.

After forming the temporary collection the search conditions of the WHERE clause will be applied to R and this will yield a new temporary collection R1. The ORDER BY and SELECT clauses are applied to R1 to yield the final result set.

An identification variable is a variable declared in the FROM clause using the operator IN or the optional AS.

```
FROM DeptBean AS d,  IN (d.emps) AS e
```

is equivalent to:

```
FROM DeptBean d, IN (d.emps) e
```

An identification variable that is declared to be an abstract schema name is called a range variable. In the query above ″d″ is a range variable. An identification variable that is declared to be a multivalued path expression is called a collection member declaration. ″d″ and ″e″ in the example above are collection member declarations.

Note that the following path expression is illegal as a collection member declaration because it is not multivalued:

```
e.dept.mgr
```

Inheritance in EJB query

If an EJB inheritance hierarchy has been defined for an abstract schema, using the abstract schema name in a query statement implies the collection of objects for that abstract schema as well as all subtypes.

Example: Inheritance

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy. The following query returns employees as well as managers and executives:

```
SELECT OBJECT(e) FROM EmpBean e
```

Path expressions

An identification variable followed by the navigation operator (.) and a cmp or relationship name is a path expression.

A path expression that leads to a cmr field can be further navigated if the cmr field is single-valued. If the path expression leads to a multi-valued relationship, then the path expression is terminal and cannot be further navigated. If the path expression leads to a cmp field whose type is a value object, it is possible to navigate to attributes of the value object.

Example: Value object

Assume that `address` is a cmp field for `EmpBean`, which is a value object.

```
SELECT  object(e)  FROM EmpBean e
WHERE  e.address.distance('San Jose') < 10  and e.address.zip = 95037
```

It is best to use the composer pattern to map value object attributes to relational columns if you intend to search on value attributes. If you store value objects in serialized format, then each value object must be retrieved from the database and deserialized. Value object methods can only be done in dynamic queries.

A path expression can also navigate to a bean method. The method must be defined on either the remote or local bean interface. Methods can only be used in dynamic queries. You cannot mix both remote and local methods in a single query statement.

If the query contains remote methods, the dynamic query must be executed using the query remote interface. Using the query remote interface causes the query service to activate beans and create instances of the remote bean interface

Likewise, a query statement with local bean methods must be executed with the query local interface. This causes the query service to activate beans and local interface instances.

Do not use get methods to access cmp and cmr fields of a bean.

If a method has overloaded definitions, the overloaded methods must have different number of parameters.

Methods must have non-void return types and method arguments and return types must be either primitive types byte, short, int, long, float, double, boolean, char or wrapper types from the following list:

Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Date

If any input argument to a method is NULL, it is assumed the method returns a NULL value and the method is not invoked.

A collection valued path expression can be used in the FROM clause as a collection member declaration, and with the IS EMPTY, MEMBER OF, and EXISTS predicates in the WHERE clause.

`FROM EmpBean e WHERE e.dept.mgr.name='Bob'`	OK
`FROM EmpBean e WHERE e.dept.emps.name='BOB'`	INVALID -- cannot navigate through emps because it is multivalued
`FROM EmpBean e,  IN (e.dept.emps) e1` `WHERE e1.name='BOB'`	OK
`FROM EmpBean e WHERE e.dept.emps IS EMPTY`	OK

WHERE clause

The WHERE clause contains search conditions composed of the following:
- literal values
- input parameters
- expressions
- basic predicates
- quantified predicates
- BETWEEN predicate
- IN predicate
- LIKE predicate
- NULL predicate
- EMPTY collection predicate
- MEMBER OF predicate
- EXISTS predicate
- IS OF TYPE predicate

If the search condition evaluates to TRUE, the tuple is added to the result set.

Literals

A string literal is enclosed in single quotes. A single quote that occurs within a string literal is represented by two single quotes; For example: 'Tom''s'. A string literal cannot exceed the maximum length that is supported by the underlying persistent datastore.

A numeric literal can be any of the following:
- an exact value such as 57, -957, +66
- any value supported by Java long
- a decimal literal such as 57.5, -47.02
- an approximate numeric value such as 7E3, -57.4E-2

A decimal or approximate numeric value must be in the range supported by the underlying persistent datastore.

A boolean literal can be the keyword TRUE or FALSE and is case insensitive.

Input parameters

Input parameters are designated by the question mark followed by a number; For example: ?2

Input parameters are numbered starting at 1 and correspond to the arguments of the finder or select method; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

An input parameter can be a primitive type of byte, short, int, long, float, double, boolean, char or wrapper types of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp or an EJBObject.

An input parameter must not have a NULL value. To search for the occurrence of a NULL value the NULL predicate should be used.

Expressions

Conditional expressions can consist of comparison operators and logical operators (AND, OR, NOT).

Arithmetic expressions can be used in comparison expressions and can be composed of arithmetic operations and functions, path expressions that evaluate to a numeric value and numeric literals and numeric input parameters.

String expressions can be used in comparison expressions and can be composed of string functions, path expressions that evaluate to a string value and string literals and string input parameters. A cmp field of type char is handled as if it were a string of length 1.

Boolean expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a boolean value and TRUE and FALSE keywords and boolean input parameters.

Reference expressions can be used with = and <> comparison and can be composed of path expressions that evaluate to a cmr field, an identification variable and an input parameter whose type is an EJB reference

Four different expression types are supported for working with date-time types. For portability the java.util.Calendar type should be used. DB2 style date, time and timestamp expressions are supported if the datastore is DB2 and the CMP field is of type java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp.

A Calendar type can be compared to another Calendar type, an exact numeric literal or input parameter of type long whose value is the standard Java long millisecond value.

The following query finds all employees born before Jan 1, 1990:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e.birthDate <  631180800232
```

Date expressions can be used in comparison expressions and can be composed of operators + - , date duration expressions and date functions, path expressions that evaluate to a date value, string representation of a date and date input parameters.

Time expressions can be used in comparison expressions and can be composed of operators + - , time duration expressions and time functions, path expressions that evaluate to a time value, string representation of time and time input parameters.

Timestamp expressions can be used in comparison expressions and can be composed of operators + - , timestamp duration expressions and timestamp functions, path expressions that evaluate to a timestamp value, string representation of a timestamp and timestamp input parameters.

Standard bracketing ( ) for ordering expression evaluation is supported.

The operators and their precedence order from highest to lowest are:
- Navigation operator (.)
- Arithmetic operators in precedence order:
  - + - unary
  - * / multiply, divide
  - + - add, subtract
- Comparison operators: =, >, <, >=, <=, <>(not equal)
- Logical operator NOT
- Logical operator AND
- Logical operator OR

In some datastores, a zero length string value (’’) is treated as a null value and affects the results of queries. Some datastores perform division between two integer values using integer arithmetic rules and other datastores use non integer rules. This also can affect the results of queries. For portability, avoid the use of zero length string values and division of integer values in an EJB query.

Null value semantics: The following describe the semantics of NULL values:
- Comparison or arithmetic operations with an unknown (NULL) value yield an unknown value
- Path expressions that contain NULL evaluate to NULL
- The IS NULL and IS NOT NULL operators can be applied to path expressions and return TRUE or FALSE. Boolean operators AND, OR and NOT use three valued logic.

AND	**True**	**False**	**Unknown**
True	True	False	Unknown
False	False	False	False
Unknown	Unknown	False	Unknown

OR	**True**	**False**	**Unknown**
True	True	True	True
False	True	False	Unknown
Unknown	True	Unknown	Unknown

	NOT
True	False
False	True
Unknown	Unknown

Example: Null value semantics

```
select object(e) from EmpBean where e.salary > 10  and e.dept.budget > 100
```

If salary is NULL the evaluation of `e.salary > 10` returns unknown and the employee object is not returned. If the cmr field dept or budget is NULL evalution of `e.dept.budget > 100` returns unknown and the employee object is not returned.

```
select object(e) from EmpBean where e.dept.budget is null
```

If dept or budget is NULL evaluation of `e.dept.budget` is null returns TRUE and the employee object is returned.

```
select object(e) from EmpBean e ,  in (e.dept.emps) e1 where e1.salary > 10
```

If dept is NULL, then the multivalued path expression `e.dept.emps` results in an empty collection (not a collection that contains a NULL value). An employee with a null dept value will not be returned.

```
select object(e) from EmpBean e where  e.dept.emps is empty
```

If dept is NULL the evaluation of the predicate in unknown and the employee object is not returned.

```
select object(e) from EmpBean e , EmpBean e1  where e member of  e1.dept.emps
```

If dept is NULL evaluation of the member of predicate returns unknown and the employee is not returned.

Date time arithmetic and comparisons: DATE, TIME and TIMESTAMP values may be compared with another value of the same type. Comparisons are chronological. Date time values can also be incremented, decremented, and subtracted.

If the datastore is DB2, then DB2 string representation of DATE, TIME and TIMESTAMP types can also be used. A string representation of a date or time can use ISO, USA, EUR or JIS format. A string representation of a timestamp uses ISO format.

Format	Date format	Date examples	Time format	Time examples
ISO	yyyy-mm-dd	1987-02-24 1987-2-24	hh.mm.ss	13.50.00 13.50
USA	mm/dd/yyyy	2/24/1987	hh:mm AM or PM	1:50 pm 02:10 AM
EUR	dd.mm.yyyy	24.02.1987 24.2.1987	hh.mm.ss	13.50.00 13.55
JIS	yyyy-mm-dd	1987-02-24	hh:mm:ss	13:50 13:50:05

Example 1: Date time arithmetic comparisons

```
e.hiredate > '1990-02-24'
```

The timestamp of February 24th, 1990 1:50 pm can be represented as follows:

```
'1990-02-24-13.50.00.000000'  or
'1990-02-24-13.50.00'
```

If the datastore is DB2, DB2 decimal durations can be used in expressions and comparisons. A date duration is a decimal(8,0) number that represents the difference between two dates in the format YYYYMMDD. A time duration is a decimal(6,0) number that represents the difference between two time values as HHMMSS. A timestamp duration is a decimal(20,6) number representing the differences between two timestamp values as YYYYMMDDHHMMSS.ZZZZZZ (ZZZZZZ is the number of microseconds and is to the right of the decimal point) .

Two date values (or time values or timestamp values) can be subtracted to yield a duration. If the second operand is greater than the first the duration is a negative decimal number. A duration can be added or subtracted from a datetime value to yield a new datetime value.

Example 2: Date time arithmetic comparisons

`DATE('3/15/2000') - '12/31/1999'` results in a decimal number 215 which is a duration of 0 years, 2 months and 15 days.

Durations are really decimal numbers and can be used in arithmetic expressions and comparisons.

`( DATE('3/15/2000') - '12/31/1999' ) + 14 > 215` evaluates to TRUE.

`DATE('12/31/1999') + DECIMAL(215,8,0)` results in a date value 3/15/2000.

`TIME('11:02:26') - '00:32:56'` results in a decimal number 102930 which is a time duration of 10 hours, 29 minutes and 30 seconds.

`TIME('00:32:56') + DECIMAL(102930,6,0)` results in a time value of 11:02:26.

`TIME('00:00:59') + DECIMAL(240000,6,0)` results in a time value of 00:00:59.

`e.hiredate + DECIMAL(500,8,0) > '2000-10-01'` means compare the hiredate plus 5 months to the date 10/01/2000.

Basic predicates

Basic predicates can be of two forms

```
expression-1  comparison-operator  expression-2
expression-3  comparison-operator ( subselect )
```

The subselect must not return more than one value and the subselect can not return a type of an EJB reference. Boolean types and reference types only support = and <> comparisons.

Example: Basic predicates

```
d.name='Java Development'
e.salary > 20000
e.salary > ( select avg(e.salary) from EmpBean e)
```

Quantified predicates

A quantified predicate compares a value with a set of values produced by a subselect.

```
expression   comparison-operator   SOME  |  ANY  |   ALL    ( subselect )
```

The expression must not evaluate to a reference type.

When SOME or ANY is specified the result of the predicate is as follows:
- TRUE if the comparison is true for at least one value returned by the subselect.
- FALSE if the subselect is empty or if the comparison is false for every value returned by the subselect.
- UNKNOWN if the comparison is not true for all of the values returned by the subselect and at least one of the comparisons is unknown because of a null value.

When ALL is specified the result of the predicate is as follows:
- TRUE if the subselect returns empty or if the comparison is true to every value returned by the subselect.
- FALSE if the comparison is false for at least one value returned by the subselect.
- UNKNOWN if the comparison is not false for all values returned by the subselect and at least one comparison is unknown because of a null value.

BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values.

```
expression  [NOT]  BETWEEN expression-2  AND expression-3
```

Example: BETWEEN predicate

```
e.salary BETWEEN 50000 AND 60000
```

is equivalent to:

```
e.salary >= 50000  AND e.salary <= 60000
e.name NOT BETWEEN 'A' AND 'B'
```

is equivalent to:

```
e.name < 'A'  OR e.name > 'B'
```

IN predicate

The IN predicate compares a value to a set of values and can have one of two forms:

```
expression [NOT] IN  ( subselect )
expression [NOT] IN  ( value1, value2,  .... )
```

ValueN can either be a literal value or an input parameter. The expression can not evaluate to a reference type.

Example: IN predicate

```
e.salary IN ( 10000, 15000 )
```

is equivalent to

```
( e.salary = 10000  OR  e.salary = 15000 )
e.salary IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary = ANY   ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
e.salary NOT IN ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary <> ALL    ( select  e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

LIKE predicate

The LIKE predicate searches a string value for a certain pattern.

```
string-expression   [NOT]  LIKE  pattern    [ ESCAPE escape-character ]
```

The pattern value is a string literal or parameter marker of type string in which the underscore (_) stands for any single character and percent (%) stands for any sequence of characters (including empty sequence). Any other character stands for itself. The escape character can be used to search for character _ and %. The escape character can be specified as a string literal or an input parameter.

If the string-expression is null, then the result is unknown.

If both string-expression and pattern are empty, then the result is true.

Example: LIKE predicate

- '' LIKE '' is true
- '' LIKE '%' is true
- e.name LIKE '12%3' is true for '123' '12993' and false for '1234'
- e.name LIKE 's_me' is true for 'some' and 'same', false for 'soome'
- e.name LIKE '/_foo' escape '/' is true for '_foo', false for 'afoo'
- e.name LIKE '//_foo' escape '/' is true for '/afoo' and for '/bfoo'
- e.name LIKE '///_foo' escape '/' is true for '/_foo' but false for '/afoo'

NULL predicate

The NULL predicate tests for null values.

```
single-valued-path-expression IS [NOT] NULL
```

Example: NULL predicate

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

EMPTY collection predicate

To test if a multivalued relationship is empty, use the following syntax:

```
collection-valued-path-expression  IS [NOT]  EMPTY
```

Example: Empty collection predicate

To find all departments with no employees:

```
SELECT OBJECT(d) FROM DeptBean d  WHERE  d.emps IS EMPTY
```

MEMBER OF predicate

This expression tests whether the object reference specified by the single valued path expression or input parameter is a member of the designated collection. If the collection valued path expression designates an empty collection the value of the MEMBER OF expression is FALSE.

```
{single-valued-path-expression | input_parameter} [NOT] MEMBER [OF] collection-valued-path-expression
```

Example: MEMBER OF predicate

Find employees that are not members of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

Find employees whose manager is a member of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d
WHERE e.dept.mgr MEMBER  OF d.emps  and d.deptno=?1
```

EXISTS predicate

The exists predicate tests for the presence or absence of a condition specified by a subselect.

```
EXISTS ( subselect )
```

```
EXISTS collection-valued-path-expression
```

The result of EXISTS is true if the subselect returns at least one value or the path expression evaluates to a nonempty collection, otherwise the result is false.

To negate an EXISTS predicate, precede it with the logical operator NOT.

Example: EXISTS predicate

Return departments that have at least one employee earning more than 1000000:

```
SELECT  OBJECT(d) FROM  DeptBean d
WHERE EXISTS ( SELECT  1  FROM IN (d.emps) e WHERE  e.salary > 1000000 )
```

Return departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d
WHERE NOT EXISTS  ( SELECT 1 FROM IN (d.emps) e)
```

The above query can also be written as follows:

```
SELECT OBJECT(d) FROM DeptBean d WHERE NOT EXISTS d.emps
```

IS OF TYPE predicate

The IS OF TYPE predicate is used to test the type of an EJB reference. It is similar in function to the Java instance of operator. IS OF TYPE is used when several abstract beans have been grouped into an EJB inheritance hierarchy. The type names specified in the predicate are the bean abstract names. The ONLY option can be used to specify that the reference must be exactly this type and not a subtype.

```
identification-variable IS OF TYPE ( [ONLY] type-1, [ONLY] type-2, ..... )
```

Example: IS OF TYPE predicate

Suppose that bean `ManagerBean` is defined as a subtype of `EmpBean` and `ExecutiveBean` is a subtype of `ManagerBean` in an EJB inheritance hierarchy.

The following query returns employees as well as managers and executives:

```
SELECT  OBJECT(e) FROM EmpBean e
```

If you are interested in objects which are employees and not managers and not executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ONLY EmpBean )
```

If you are interested in object which are managers or executives:

```
SELECT OBJECT(e) FROM EmpBean e WHERE e IS OF TYPE( ManagerBean)
```

The above query is equivalent to the following query:

```
SELECT  OBJECT(e) FROM ManagerBean e
```

If you are interested in managers only and not executives:

```
SELECT  OBJECT(e) FROM  EmpBean e WHERE e IS OF TYPE( ONLY ManagerBean)
```

or:

```
SELECT  OBJECT(e) FROM  ManagerBean e
WHERE  e IS OF TYPE (ONLY ManagerBean)
```

Scalar functions

EJB query contains scalar built-in functions for doing type conversions, string manipulation, and for manipulating date-time values. The list of scalar functions is documented in the topic EJB query: Scalar functions.

Example: Scalar functions

Find employees hired in 1999:

```
SELECT OBJECT(e) FROM EmpBean e where YEAR(e.hireDate) = 1999
```

The only scalar functions that are guaranteed to be portable across backend datastore vendors are the following:
- ABS
- SQRT
- CONCAT
- LENGTH
- LOCATE
- SUBSTRING

The other scalar functions should be used only when DB2 is the backend datastore.

EJB query: Scalar functions

EJB query contains scalar built-in functions, as listed below, for doing type conversions, string manipulation, and for manipulating date-time values.

Numeric functions

```
ABS ( < any numeric datatype > ) -> < any numeric datatype >
```

```
SQRT ( < any numeric datatype > ) -> Double
```

Type conversion functions

```
CHAR ( < any  numeric datatype > ) ->  string
CHAR ( <  string  > ) ->  string
CHAR ( < any datetime  datatype >  [, Keyword k ]) ->  string
```

Datetime datatype is converted to its string representation in a format specified by the keyword k. The valid keywords values are ISO, USA, EUR or JIS. If k is not specified the default is ISO.

```
BIGINT ( < any numeric datatype > ) -> Long
BIGINT ( < string > ) -> Long
```

The following function converts the argument to an integer n by truncation and returns the date that is n-1 days after January 1, 0001:

```
DATE ( < date string > ) -> Date
DATE (  < any numeric datatype>) -> Date
```

The following function returns date portion of a timestamp:

```
DATE( timestamp ) -> Date
DATE ( < timestamp-string > ) -> Date
```

The following function converts number to decimal with optional precision p and scale s.

```
DECIMAL ( < any numeric datatype > [, p [ ,s ] ] ) -> Decimal
```

The following function converts string to decimal with optional precision p and scale s.

```
DECIMAL ( < string > [ , p [ , s ] ] ) -> Decimal
```

```
DOUBLE ( < any numeric datatype > ) -> Double
DOUBLE ( < string > ) -> Double
```

```
FLOAT ( < any numeric datatype > ) -> Double
FLOAT ( < string > ) -> Double
```

Float is a synonym for DOUBLE.

```
INTEGER ( < any numeric datatype > ) -> Integer
INTEGER ( < string > ) -> Integer
```

```
REAL ( < any numeric datatype > ) -> Float
```

```
SMALLINT ( < any numeric datatype ) -> Short
SMALLINT ( < string > ) -> Short
```

```
TIME ( < time > ) -> Time
TIME ( < time-string  > ) -> Time
TIME ( < timestamp > ) -> Time
TIME ( < timestamp-string  > ) -> Time
```

```
TIMESTAMP ( < timestamp > ) -> Timestamp
TIMESTAMP ( < timestamp-string > ) -> Timestamp
```

String functions

```
CONCAT ( <string>, <string>  ) -> String
```

The following function returns a character string representing absolute value of the argument not including its sign or decimal point. For example, `digits( -42.35)` is "4235".

```
DIGITS ( Decimal d  ) -> String
```

The following function returns the length of the argument in bytes. If the argument is a numeric or datetime type, it returns the length of internal representation.

```
LENGTH ( < string >  ) -> Integer
```

The following function returns a copy of the argument string where all upper case characters have been converted to lower case.

```
LCASE ( < string > ) -> String
```

The following function returns the starting position of the first occurrence of argument 1 inside argument 2 with optional start position. If not found, it returns 0.

```
LOCATE ( String s1 , String s2  [, Integer start ] ) -> Integer
```

The following function returns a substring of s beginning at character m and containing n characters. If n is omitted, the substring contains the remainder of string s. The result string is padded with blanks if needed to make a string of length n.

```
SUBSTRING ( String s ,  Integer m [ , Integer n ] ) -> String
```

The following function returns a copy of the argument string where all lower case characters have been converted to upper case.

```
UCASE ( < string > ) -> String
```

Date - time functions

The following function returns the day portion of its argument. For a duration, the return value can be -99 to 99.

```
DAY (  Date ) ->  Integer
DAY ( < date-string > ) ->  Integer
DAY ( < date-duration > ) -> Integer
DAY ( Timestamp  ) ->  Integer
DAY ( < timestamp-string > ) ->  Integer
DAY ( < timestamp-duration > ) -> Integer
```

The following function returns one more than number of days from January 1, 0001 to its argument.

```
DAYS ( Date  ) ->  Integer
DAYS ( < Date-string > ) ->  Integer
DAYS ( Timestamp  ) ->  Integer
DAYS ( < timestamp-string > ) ->  Integer
```

The following function returns the hour part of its argument. For a duration, the return value can be -99 to 99.

```
HOUR ( Time ) -> Integer
HOUR ( < time-string > ) -> Integer
HOUR ( < time-duration > ) ->  Integer
HOUR ( Timestamp ) -> Integer
HOUR ( < timestamp-string > ) -> Integer
HOUR ( < timestamp-duration >  ) -> Integer
```

The following function returns the microsecond part of its argument.

```
MICROSECOND ( Timestamp ) -> Integer
MICROSECOND ( < timestamp-string > ) -> Integer
MICROSECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the minute part of its argument. For a duration, the return value can be -99 to 99.

```
MINUTE ( Time ) -> Integer
MINUTE ( < time-string > ) -> Integer
MINUTE ( < time-duration > ) ->  Integer
MINUTE ( Timestamp ) -> Integer
MINUTE ( < timestamp-string > ) -> Integer
MINUTE ( < timestamp-duration >  ) -> Integer
```

The following function returns the month portion of its argument. For a duration, the return value can be -99 to 99.

```
MONTH (  Date ) ->  Integer
MONTH ( < date-string > ) ->  Integer
MONTH ( < date-duration > ) -> Integer
MONTH ( Timestamp  ) ->  Integer
MONTH ( < timestamp-string > ) ->  Integer
MONTH ( < timestamp-duration > ) -> Integer
```

The following function returns the second part of its argument. For a duration, the return value can be -99 to 99.

```
SECOND ( Time ) -> Integer
SECOND ( < time-string > ) -> Integer
SECOND ( < time-duration > ) ->  Integer
SECOND ( Timestamp ) -> Integer
SECOND ( < timestamp-string > ) -> Integer
SECOND ( < timestamp-duration >  ) -> Integer
```

The following function returns the year portion of its argument. For a duration, the return value can be -9999 to 9999.

```
YEAR (  Date ) ->  Integer
YEAR ( < date-string > ) ->  Integer
YEAR ( < date-duration > ) -> Integer
YEAR ( Timestamp  ) ->  Integer
YEAR ( < timestamp-string > ) ->  Integer
YEAR ( < timestamp-duration > ) -> Integer
```

Aggregation functions

Queries that return aggregate values can only be used with the dynamic query interface available in WebSphere Application Server Enterprise. However, aggregation functions can be used in non-dynamic queries if the aggregation function is used in a subselect or HAVING clause.

Aggregation functions operate on a set of values to return a single scalar value. The following is an example of an aggregation:

```
SELECT  SUM (e.salary + e.bonus) FROM EmpBean e WHERE e.dept.deptno =20
```

This computes the total salary and bonus for department 20.

The aggregation functions are avg, count, max, min and sum. The syntax of an aggregation function is as follows:

```
aggregation-function  (    [ ALL |  DISTINCT ]  expression )
```

or:

```
COUNT( * )
```

The DISTINCT option eliminates duplicate values before applying the function. ALL is the default and does not eliminate duplicates. Null values are ignored in computing the aggregate function except for COUNT(*) which returns a count of all elements in the set.

MAX and MIN can apply to any numeric, string or datetime datatype and returns the same datatype. SUM and AVG take a numeric type as input. SUM and AVG return numeric type. The actual numeric type returned by SUM and AVG depends on the underlying datastore. COUNT can take any datatype and returns an integer.

If you are using an Informix datastore, the argument to COUNT must be an asterisk or a single valued path expression. The argument to SUM, AVG, MIN, or MAX used with DISTINCT must be a single valued path expression.

The set of values that is used for the aggregate function is determined by the collection that results from the FROM and WHERE clause of the subquery. This set can be divided into groups and the aggregation function applied to each group. This is done by using a GROUP BY clause in the subquery. The GROUP BY clause defines grouping members which is a list of path expressions. Each path

expression specifies a field that is a primitive type of byte, short, int, long, float, double, boolean, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time or java.sql.Timestamp.

Finder or select queries can not return aggregation function values. In other words, aggregation functions can not appear in the top level SELECT of a finder or select query but can be used in subqueries.

Example: Aggregation functions

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

The above query computes the average salary for each department.

In dividing a set into groups, a NULL value is considered equal to another NULL value.

Just as the WHERE clause filters tuples from the FROM clause, the groups can be filtered using a HAVING clause that tests group properties involving aggregate functions or grouping members:

```
SELECT e.dept.deptno,  AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING  COUNT(*) > 3  AND  e.dept.deptno > 5
```

This query returns average salary for departments that have more than 3 employees and the department number is greater than 5.

It is possible to have a HAVING clause without a GROUP BY clause in which case the entire set is treated as a single group to which the HAVING clause is applied.

SELECT clause

For finder and select queries, the syntax of the SELECT clause is as follows:

```
SELECT  [  ALL | DISTINCT  ]
 { single-valued-path-expression  |  OBJECT ( identification-variable )  }
```

The SELECT clause consists of either a single identification variable that is defined in the FROM clause or a single valued path expression that evaluates to a object reference or CMP value. The keyword DISTINCT can be used to eliminate duplicate references.

For a query that defines a finder method the query must return an object type consistent with the home for which the finder method associated with the query. In other words, a finder method for a department home can not return employee objects.

A scalar-subselect is a subselect that returns a single value.

Example: SELECT clause

Find all employees that earn more than John:

```
SELECT OBJECT(e) FROM EmpBean ej, EmpBean e
WHERE  ej.name = 'John'  and e.salary > ej.salary
```

Find all departments that have one or more employees who earn less than 20000:

```
SELECT DISTINCT  e.dept  FROM EmpBean e where e.salary < 20000
```

A select method query can have a path expression that evaluates to an arbitrary value:

```
SELECT  e.dept.name  FROM EmpBean e where e.salary < 2000
```

The above query returns a collection of name values for those departments having employees earning less than 20000.

ORDER BY clause

The ORDER BY clause specifies an ordering of the objects in the result collection:

```
ORDER BY  [ order_element ,]* order_element
order_element ::= { path-expression | integer } [ ASC | DESC ]
```

The path expression must specify a single valued field that is a primitive type of byte, short, int, long, float, double, char or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Character, java.util.Calendar, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp.

ASC specifies ascending order and is the default. DESC specifies descending order.

Integer refers to a selection expression in the SELECT clause.

Example: ORDER BY clause

Return department objects in decreasing deptno order:

```
SELECT  OBJECT(d) FROM  DeptBean d ORDER BY  d.deptno DESC
```

Return employee objects sorted by department number and name:

```
SELECT  OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC,  e.name DESC
```

Subqueries

A subquery can be used in quantified predicates, EXISTS predicate or IN predicate. A subquery should only specify a single element in the SELECT clause. When a path expression appears in a subquery, the identification variable of the path expression must be defined either in the subquery, in one of the containing subqueries, or in the outer query. A scalar subquery is a subquery that returns one value. A scalar subquery can be used in a basic predicate and in the SELECT clause of a dynamic query.

Example: Subqueries

```
SELECT  OBJECT(e) FROM  EmpBean e
WHERE e.salary > ( SELECT  AVG(e1.salary) FROM  EmpBean e1)
```

The above query returns employees who earn more than average salary of all employees.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE  e.salary >
( SELECT AVG(e1.salary) FROM  IN  (e.dept.emps) e1  )
```

The above query returns employees who earn more than average salary of their department.

```
SELECT  OBJECT(e) FROM EmpBean e WHERE e.salary =
( SELECT MAX(e1.salary) FROM IN (e.dept.emps) e1  )
```

The above query returns employees who earn the most in their department.

```
 SELECT OBJECT(e) FROM EmpBean e
WHERE e.salary > ( SELECT AVG(e.salary) FROM EmpBean e1
WHERE YEAR(e1.hireDate) =  YEAR(e.hireDate)  )
```

The above query returns employees who earn more than the average of employees hired in same year.

EJB query restrictions

An EJB query is compiled into an SQL query and executed against the underlying datastore based on schema mapping of the abstract bean to the datastore schema. The semantics of comparison and arithmetic operations are that of the underlying datastore. In the case of SQL, note that two strings are equal if the shorter string padded with blanks equals the longer string. For example, 'A' is equal to 'A '. This differs from the equality of strings in the Java language. Arithmetic overflow operations are an error in SQL.

A cmp field can not be used in comparison operations or predicates (except for LIKE) if that cmp field is mapped to a long varchar or large objects (LOB) column or any other column type for which the database server does not support predicates or comparison operations.

A cmp field of any type can be used in a SELECT clause. Fields that can be used in predicates, grouping, or ordering operations must be of the types listed below:
- Primitive types : byte, short, int, long, float, double, boolean, char
- Object types: Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Char, java.util.Calendar , java.util.Date
- JDBC types: java.sql.Date, java.sql.Time, java.sql.Timestamp

The field must be mapped to a table column that is compatible in type either by using a ″top-down″ default mapping generated by the WebSphere deploy tool, or using a ″meet-in-the-middle″ mapping between compatible types.

In order to search on attributes of a cmp field that is a user-defined value object, you should use a ″meet-in-the-middle″ mapping and use a composer to map each attribute to a compatible column. The default ″top-down″ mapping stores the object as a serialized object in a column of type blob, which does not allow searching.

If a cmp field is mapped to a column using a ″meet-in-the-middle″ mapping with a converter, that field can only be used with the NULL predicate or with basic predicates of the following form:

```
 path-expression  <comparison>  literal_value
 path-expression  <comparison>  input_parameter
```

In this situation, the converter method toData( ) is called to convert the right-hand side of the predicate to an SQL value.

Example of allowable predicate on a cmp field with user defined converter:

```
e.name = 'Chris'
e.name > ?1
e.name IS NULL
```

Examples of unallowable predicates:

```
substring( e.name, 1, 3 ) = 'ABC'
e.salary >  d.budget
```

A converter should preserve equality, collating sequence and null values when doing a conversion. Otherwise cmp fields created by the converter should not be used in WHERE, GROUP, HAVING or ORDER clauses of a query.

EJB Query: Reserved words

The following words are reserved in WebSphere EJB query:

all, as, distinct, empty, false, from, group, having, in, is, like, select, true, union, where

Avoid using identifiers that start with underscore (for example, _integer) as these are also reserved.

EJB query: BNF syntax

```
EJB QL ::= [select_clause] from_clause [where_clause]
          [group_by_clause] [having_clause] [order_by_clause]

from_clause::=FROM identification_variable_declaration
              [,identification_variable_declaration]*

identification_variable_declaration::=collection_member_declaration |
     range_variable_declaration

collection_member_declaration::=
     IN ( collection_valued_path_expression ) [AS] identifier

range_variable_declaration::=abstract_schema_name [AS] identifier

single_valued_path_expression ::=
      {single_valued_navigation | identification_variable}. ( cmp_field |
             method  |  cmp_field.value_object_attribute |  cmp_field.value_object_method )
       | single_valued_navigation

single_valued_navigation::=
       identification_variable.[  single_valued_cmr_field.  ]*
            single_valued_cmr_field

collection_valued_path_expression ::=
       identification_variable.[  single_valued_cmr_field.  ]*
           collection_valued_cmr_field

select_clause::= SELECT { ALL | DISTINCT }  {single_valued_path_expression |
                         identification_variable |  OBJECT ( identification_variable) }

select_clause_eex ::= SELECT { ALL | DISTINCT }  [ selection , ]*  selection

selection  ::= { expression [[AS] id ]  |  subselect }

order_by_clause::= ORDER BY [ {single_valued_path_expression | integer}  [ASC|DESC],]*
      {single_valued_path_expression | integer}[ASC|DESC]

where_clause::= WHERE conditional_expression

conditional_expression ::= conditional_term |
                           conditional_expression OR conditional_term

conditional_term ::= conditional_factor |
                     conditional_term AND conditional_factor

conditional_factor ::= [NOT] conditional_primary

conditional_primary::=simple_cond_expression | (conditional_expression)

simple_cond_expression ::= comparison_expression | between_expression |
       like_expression | in_expression | null_comparison_expression |
       empty_collection_comparison_expression | quantified_expression |
       exists_expression | is_of_type_expression  |  collection_member_expression

between_expression ::= expression [NOT] BETWEEN expression AND expression

in_expression ::= single_valued_path_expression [NOT] IN
           { (subselect) |  ( atom ,]* atom) }

atom = { string-literal | numeric-constant | input-parameter }
```

```
like_expression ::= expression [NOT] LIKE
                {string_literal | input_parameter}
                [ESCAPE {string_literal | input_parameter}]

null_comparison_expression ::=
       single_valued_path_expression IS [ NOT ] NULL

empty_collection_comparison_expression ::=
       collection_valued_path_expression IS [NOT] EMPTY

collection_member_expression  ::=
       { single_valued_path_expression |  input_paramter }  [ NOT ] MEMBER [ OF ]
         collection_valued_path_expression

quantified_expression ::=
       expression comparison_operator  {SOME | ANY | ALL} (subselect)

exists_expression ::= EXISTS {collection_valued_path_expression |  (subselect)}

subselect ::= SELECT [{ ALL | DISTINCT }]  expression  from_clause [where_clause]
       [group_by_clause] [having_clause]

group_by_clause::= GROUP BY [single_valued_path_expression,]*
                    single_valued_path_expression

having_clause ::= HAVING conditional_expression

is_of_type_expression ::= identifier  IS OF TYPE
            ([[ONLY] abstract_schema_name,]* [ONLY] abstract_schema_name)

comparison_expression ::=  expression   comparison_operator { expression |  ( subquery ) }

comparison_operator ::=    = | > | >= | < | <= | <>

method ::=  method_name( [[expression , ]* expression ] )

expression ::= term |   expression {+|-} term

term ::=  factor |  term {*|/} factor

factor ::= {+|-} primary

primary ::= single_valued_path_expression | literal |
       ( expression ) |  input_parameter | functions

functions ::=
        ABS(expression) |
        AVG([ALL|DISTINCT] expression) |
        BIGINT(expression) |
        CHAR({expression [,{ISO|USA|EUR|JIS}] )  |
        CONCAT (expression , expression ) |
        COUNT({[ALL|DISTINCT] expression | *}) |
        DATE(expression) |
        DAY({expression ) |
        DAYS( expression) |
        DECIMAL( expression [,integer[,integer]])
        DIGITS( expression) |
        DOUBLE( expression ) |
        FLOAT( expression) |
        HOUR ( expression ) |
        INTEGER( expression ) |
        LCASE ( expression) |
        LENGTH(expression) |
        LOCATE( expression, expression [, expression] ) |
        MAX([ALL|DISTINCT] expression) |
        MICROSECOND( expression ) |
        MIN([ALL|DISTINCT] expression) |
        MINUTE ( expression ) |
        MONTH( expression ) |
        REAL( expression) |
        SECOND( expression ) |
        SMALLINT( expression )  |
        SQRT (  expression) |
        SUBSTRING( expression, expression[, expression]) |
        SUM([ALL|DISTINCT] expression) |
```

```
TIME( expression ) |
TIMESTAMP( expression ) |
UCASE ( expression) |
YEAR( expression )
```

Comparison of EJB 2.0 specification and WebSphere query language

Item	EJB 2.0 specification	WebSphere Query	WebSphere Enterprise (Dynamic) Query
Bean methods	no	no	yes
Calendar comparisons	yes	yes	yes
Delimited identifiers	no	yes	yes
Dependent Value attributes	no	yes	yes
Dependent Value methods	no	no	yes
Dynamic Query APIs	no	no	yes
EXISTS predicate	no	yes	yes
Inheritance	no	yes	yes
Multiple element select clauses	no	no	yes
Order by	no	yes	yes
Scalar functions	yes *	yes	yes
Select clause	required	optional	optional
SQL Date/time expressions	no	yes	yes
String comparisons	= and <> only	= <> > <	= <> > <
Subqueries, aggregations, group by, and having clauses	no	yes	yes

* EJB 2.0 defines the following scalar functions: abs, sqrt, concat, length, locate and substring. WebSphere query and dynamic query support additional scalar functions as listed in the topic, EJB query: Scalar functions.

Chapter 12. Internationalizing applications

For your application to be used in multiple regions around the world, its user interfaces will need to support multiple locales and time zones. IBM WebSphere Application Server supports the maintenance and deployment of centralized message catalogs for the output of properly formatted, language-specific (*localized*) interface strings.

- If you are new to internationalization, read ″Internationalization″ before you continue.
- For general information about internationalization, see ″Resources for learning.″

1. Identify localizable text in your application.
2. Create the message catalogs necessary for the locales to be supported by your application.
3. In your application code, compose the language-specific strings for output.
4. Assemble your application code as one or more application components.
5. Prepare the localizable-text package for deployment with your localized application. In this step, you create a deployment JAR.
6. 5.0.2 + Using the Assembly Toolkit, assemble the application modules and the deployment JAR into a J2EE application.
7. 5.0.1 Using the Application Assembly Tool (AAT), assemble the application modules and the deployment JAR into a J2EE application.
8. Deploy and manage the application.

Internationalization

An application that can present information to users according to regional cultural conventions is said to be *internationalized*: The application can be configured to interact with users from different localities in culturally appropriate ways. In an internationalized application, a user in one region sees error messages, output, and interface elements in the requested language. Date and time formats, as well as currencies, are presented appropriately for users in the specified region. A user in another region sees output in the conventional language or format for that region.

Historically, the creation of internationalized applications has been restricted to large corporations writing complex systems. Internationalization techniques have traditionally been expensive and difficult to implement, so they have been applied only to major development efforts. However, given the rise in distributed computing and in the use of the World Wide Web, application developers have been pressured to internationalize a much wider variety of applications. This requires making internationalization techniques much more accessible to application developers.

In an application that is not internationalized, the interface that the user sees is unalterably written into the application code. On the other hand, localizing the displayed strings adds a layer of abstraction into the design of the application. Instead of simply printing an error message, an internationalized application represents the error message with some language-neutral information; in the simplest case, each error condition corresponds to a key. To print a usable error message, then, the application looks up the key in the configured message catalog. Each message catalog is a list of keys with associated strings. Different message catalogs provide strings for the different languages supported. The application

© Copyright IBM Corp. 2003

looks up the key in the appropriate catalog, retrieves the corresponding error message in the requested language, and prints this string for the user.

Localization of text can be used for far more than translating error messages. For example, by using keys to represent each element in a graphical user interface (GUI) and by providing the appropriate message catalogs, the GUI itself (buttons, menus, and so on) can support multiple languages. Extending support to additional languages requires that you provide message catalogs for those languages; in many cases, the application itself needs no further modification.

Internationalization of an application is driven by two variables, the time zone and the locale. The time zone indicates how to compute the local time as an offset from a standard time like Greenwich Mean Time. The locale is a collection of information about language, currency, and the conventions for presenting information like dates. In a localized application, the locale also indicates the message catalog from which an application is to retrieve message strings. A time zone can cover many locales, and a single locale can span time zones. With both time zone and locale, the date, time, currency, and language for users in a specific region can be determined.

Identifying localizable text

1. Determine which elements of the application need to be translated. Good candidates for localization include the following:
   - Graphical user interfaces: window titles, menus and menu items, buttons, on-screen instructions
   - Prompts in command-line interfaces
   - Application output: messages and logs
2. Assign a unique key to each element for use in message catalogs for the application. The key provides a language-neutral link between the application and language-specific strings in the message catalogs. Establishing a naming convention for keys before creating the catalogs can make writing code with these keys much more intuitive for interface programmers.

Suppose you are localizing the GUI for a banking system, and the first window contains a pull-down list to be used for selecting a type of account. The labels for the list and the account types in the list are good choices for localization. Three elements require keys: the list itself and two items in the list.

Create message catalogs for the language-specific strings.

Creating message catalogs

Identify strings that need to be localized.

You can create a catalog as either a subclass of java.util.ResourceBundle or a Java properties file. The properties-file approach is more common, because properties files can be prepared by people without programming experience and swapped in without modifying the application code.

1. For each string identified for localization, add a line to the message catalog that lists the string's key and value in the current language. In a properties file, each line has the following structure:

   ```
   key = string associated with the key
   ```

2. Save the catalog, giving it a locale-specific name. To enable resolution to a specific properties file, the Java API specifies naming conventions for the properties files in a resource bundle as *bundleName_localeID*.properties. Give the set of message catalogs a collective name, for example, BankingResources. For information about locale IDs that are recognized by the Java APIs, see ″Resources for learning.″

The following English catalog (BankingResources_en.properties) supports the labels for the list and its two list items:

```
accountString = Accounts
savingsString = Savings
checkingString = Checking
```

The corresponding German catalog (BankingResources_de.properties) supports the labels as follows:

```
accountString = Konten
savingsString = Sparkonto
checkingString = Girokonto
```

Write code to compose the language-specific strings.

Composing language-specific strings

Create message catalogs for the language-specific strings.

1. In application code, create a LocalizableTextFormatter instance, passing in required localization values.
2. Set other localization values as needed for more complex situations.
3. Generate a properly formatted, language-specific string.

When the application is finished, deploy your application.For more information, see “Preparing the localizable-text package for deployment” on page 758.

Localization API support

The package *com.ibm.websphere.i18n.localizabletext* contains classes and interfaces for localizing text. This package makes extensive use of the internationalization features of the standard Java APIs from Sun Microsystems, including the following:

- java.util.Locale
- java.util.TimeZone
- java.util.ResourceBundle
- java.text.MessageFormat

For more information about the standard Java APIs, see ″Resources for learning.″

The WebSphere localizable-text package wraps the Java support and extends it for efficient and simple use in a distributed environment. The primary class used by application programmers is LocalizableTextFormatter. Instances of this class are usually created in server programs, but client programs can also create them. Formatter instances are created for specific resource-bundle names and keys. Client programs that receive a LocalizableTextFormatter instance call its format method. This method uses the locale of the client application to retrieve the appropriate resource bundle and compose a locale-specific message based on the key.

For example, suppose that a distributed application supports both French and English locales; the server is using an English locale and the client, a French locale.

The server creates two resource bundles, one each for English and French. When the client makes a request that triggers a message, the server creates a LocalizableTextFormatter instance that contains the name of the resource bundle and the key for the message and passes the instance back to the client.

When the client receives the LocalizableTextFormatter instance, it calls the object's format method. By using the locale and name of the resource bundle, the format method determines the name of the resource bundle that supports the French locale and retrieves the message that corresponds to the key from the French resource bundle. Formatting of the message is transparent to the client.

In this simple example, the resource bundles reside centrally with the server. They do not have to exist with the client. Part of what the localizable-text package provides is the infrastructure to support centralized catalogs. This implementation uses an enterprise bean (a stateless session bean provided with the localizable-text package) to access the message catalogs. When the client calls the format method on the LocalizableTextFormatter instance, the following events occur:

1. The client application sets the time-zone and locale values in the LocalizableTextFormatter instance, either by passing them explicitly or through default values.
2. A call, LocalizableTextFormatterEJBFinder, is made to retrieve a reference to the formatter bean.
3. Information from the LocalizableTextFormatter instance, including the client's time zone and locale, is sent to the formatting bean.
4. The formatting bean uses the name of the resource bundle, the message key, the time zone, and the locale to compose a language-specific message.
5. The formatter bean returns the formatted message to the client.
6. The formatted message is inserted into the LocalizableTextFormatter instance and returned by the format method.

A call to the format method requires at most one remote call, to contact the formatter bean. As an alternative, the LocalizableTextFormatter instance can cache formatted messages, eliminating the remote call for subsequent uses. In addition, you can set a fallback string so that the application can return a readable string even if it cannot access the appropriate message catalog.

The resource bundles can be stored locally. The localizable-text package provides a static variable that indicates whether the bundles are stored locally (LocalizableConfiguration.LOCAL) or remotely (LocalizableConfiguration.REMOTE). However, the setting of this variable applies to all applications running within the same Java virtual machine.

LocalizableTextFormatter class

The LocalizableTextFormatter class, found in the package com.ibm.websphere.i18n.localizabletext, is the primary programming interface for using the localizable-text package. Instances of this class contain the information needed to create language-specific strings from keys and resource bundles.

LocalizableTextFormatter extends java.lang.Object and implements the following interfaces:

- java.io.Serializable
- com.ibm.websphere.i18n.localizabletext.LocalizableText
- com.ibm.websphere.i18n.localizabletext.LocalizableTextL
- com.ibm.websphere.i18n.localizabletext.LocalizableTextTZ
- com.ibm.websphere.i18n.localizabletext.LocalizableTextLTZ

Creation and initialization of class instances

LocalizableTextFormatter supports the following constructors:
- LocalizableTextFormatter()
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName)
- LocalizableTextFormatter(String resourceBundleName, String patternKey, String appName, Object[] args)

The LocalizableTextFormatter instance must have certain values, such as resource-bundle name, key, and the name of the formatting application. If you do not pass these values in by using the second constructor listed previously, you can set them separately by making the following calls:
- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)

You can use a fourth method, setArguments(Object[] args), to set optional localization values after construction. See Processing of application-specific values at the end of this article. For a usage example, see ″Composing complex strings.″

API for formatting text

The formatting methods in LocalizableTextFormatter generate a string from a set of message keys and resource bundles, based on some combination of locale and time-zone values. Each method corresponds to one of the four localizable-text interfaces implemented. The following list indicates the interface in which each formatting method is defined:
- LocalizableText.format()
- LocalizableTextL.format(java.util.Locale locale)
- LocalizableTextTZ.format(java.util.TimeZone timeZone)
- LocalizableTextLTZ.format(java.util.Locale locale, java.util.TimeZone timeZone)

The format method with no arguments uses the locale and time-zone values set as defaults for the Java virtual macine. All four methods throw LocalizableException objects.

Location of message catalogs and the appName value

Applications written with the localizable-text package can access message catalogs locally or remotely. In a distributed environment, use of remote, centrally located message catalogs is appropriate. All clients can use the same catalogs, and maintenance of the catalogs is simplified. Local formatting is useful in test situations and apppropriate under some circumstances. In order to support either local or remote formatting, a LocalizableTextFormatter instance must indicate the name of the formatting application. For example, when an application formats a message by using remote catalogs, the message is actually formatted by an enterprise bean on the server. Although the localizable-text package contains the code to automate the lookup of the formatter bean and the issue of a call to it, the application needs to know the name of the formatter bean. Several methods in the LocalizableTextFormatter class use a value described as *appName*; this refers to the name of the formatting application, which is not necessarily the name of the application in which the value is set.

Caching of messages

LocalizableTextFormatter can optionally cache formatted messages so that they do not have to be reformatted when needed again. By default, caching is not enabled, but use `LocalizableTextFormatter.setCacheSetting(true)` to enable caching. When caching is enabled and the format method is called, the method determines whether the message has already been formatted. If so, the cached message is returned. If the message is not found in the cache, the message is formatted and returned to the caller, and a copy of the message is cached for future use.

If caching is disabled after messages have been cached, those messages remain in the cache until the cache is cleared by a call to LocalizableTextformatter.clearCache(). You can clear the cache at any time; the cache is automatically cleared when any of the following methods is called:
- setResourceBundleName(String resourceBundleName)
- setPatternKey(String patternKey)
- setApplicationName(String appName)
- setArguments(Object[] args)

API for providing fallback information

Under some circumstances, it can be impossible to format a message. The localizable-text package implements a fallback strategy, making it possible to get some information even if a message cannot be formatted correctly into the requested language. The LocalizableTextFormatter instance can optionally store fallback values for a message string, the time zone, and the locale. These can be ignored unless the LocalizableTextFormatter instance throws an exception. To set fallback values, call the following methods as appropriate:
- setFallBackString(String message)
- setFallBackLocale(Locale locale)
- setFallBackTimeZone(TimeZone timeZone)

For a usage example, see ″Generating localized text.″

Processing of application-specific values

The localizable-text package provides native support for localization based on time zone and locale, but one can construct messages on the basis of other values as well. If you need to consider variables other than locale and time zone in formatting localized text, write your own formatter class.

Your formatter class can extend LocalizableTextFormatter or independently implement some or all of the same localizable-text interfaces. As a minimum, your class must implement java.io.Serializable and at least one of the localizable-text interfaces and its corresponding format method. If your class implements more than one localizable-text interface and format method, the order of evaluation of the interfaces is as follows:
1. LocalizableTextLTZ
2. LocalizableTextL
3. LocalizableTextTZ
4. LocalizableText

As an example, the localizable-text package provides a class that reports the time and date (LocalizableTextDateTimeArgument). In that class, date and time formatting is localized in accordance with three values: locale, time zone, and style.

Creating a formatter instance

Server programs typically create LocalizableTextFormatter instances that are sent to clients as the result of some operation; clients format the objects at the appropriate time. Less typically, client programs create LocalizableTextFormatter objects locally.

1. If needed for your application, write your own formatter class. For more information about implementation, see ″LocalizableTextFormatter class.″
2. In application code, call the appropriate constructor for the formatter class and set required localization values. Some localization values, such as resource bundle name, key and formatting application, must be set, either through a constructor or soon after construction. Other localization values can be set only as needed. For more information about the API, see the related reference.

The following code creates a LocalizableTextFormatter instance by using the default constructor and then sets the required localization values:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType) {
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ltf.setPatternKey("accountNumber");
   ltf.setResourceBundleName("BankingSample.BankingResources");
   ltf.setApplicationName("BankingSample");
   ...
}
```

The application that is requesting a localized message can specify the locale and time zone for which the message is to be formatted, or the application can use the default values set for the Java virtual machine.

For example, a GUI can enable users to select the language in which to display the interface. A default value must be set initially so that the GUI can be created properly when the application first starts, but users can then change the locale for the GUI to suit their needs. The following code shows how to change the locale used by an application based on the selection of a menu item:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
import java.util.Locale;

public void actionPerformed(ActionEvent event) {
   String action = event.getActionCommand();
   ...
   if (action.equals("en_us")) {
      applicationLocale = new Locale("en", "US");
      ...
   }
   if (action.equals("de_de")) {
      applicationLocale = new Locale("de", "DE");
      ...
   }
   if (action.equals("fr_fr")) {
      applicationLocale = new Locale("fr", "FR");
      ...
   }
   ...
}
```

For more information, see "Generating localized text."

Set optional localization values.

Setting optional localization values

In addition to setting localization values that are required by LocalizableTextFormatter, you can set a number of optional values in application code, either through the constructor or by calling any of several methods for that purpose. With optional values, you can do the following:
- Compose complex strings from variable substrings
- Customize the formatting of strings, taking into account variables other than time zone and locale

1. In application code, add the optional values into an array of type Object.
   ```
   Object[] arg = {new String(getAccountNumber())};
   ```
2. Pass the array into a LocalizableTextFormatter instance. You can pass the array through the appropriate constructor or by calling the setArguments(Object[]) method. For a usage example, see "Composing complex strings."

   Note: Because the array is passed by value rather than by reference, any updates to the array variable after this point are not reflected in the LocalizableTextFormatter instance unless it is reset by calling the setArguments(Object[]) method.

Write code to generate the localized text.

Composing complex strings

Identify strings that need to be localized.

The localized-text package supports the substitution of variable substrings into a localized string that is retrieved from the message catalog by key.

1. In the message catalog, specify the location of the substitution in the string to be retrieved by key. Variable components are designated by curly braces (for example, `{0}`).
2. In application code, create a LocalizableTextFormatter instance, passing in an array that contains the variable value. If the variable substring must itself be localized, you can create a nested LocalizableTextFormatter instance for it and pass the instance in instead of a value.
3. Generate a localized string. When a format method is called on a formatter instance, the formatter takes each element of the array passed in the previous step and substitutes it for the placeholder with the matching index in the string retrieved from the message catalog. For example, the value at index 0 in the array replaces the `{0}` variable in the retrieved string.

The following line from an English message catalog shows a string with a single substitution:

```
successfulTransaction = The operation on account {0} was successful.
```

The same key in message catalogs for other languages has a translation of this string with the variable at the appropriate location for each language.

The following code shows the creation of a single-element argument array and the creation and use of a LocalizableTextFormatter instance:

```
public void updateAccount(String transactionType) {
   ...
   Object[] arg = {new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
                                    "BankingSample",
                                    arg);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Nesting formatter instances for localized substrings:

Identify strings that need to be localized.

The ability to substitute variable substrings into the strings retrieved from message catalogs adds a level of flexibility to the localizable-text package, but this capability is of limited use unless the variable value itself can be localized. You can do this by nesting LocalizableTextFormatter instances.

1. In the message catalog, add entries that correspond to potential values for the variable substring.
2. In application code, create a LocalizableTextFormatter instance for the variable substring, setting required localization values.
3. Create a LocalizableTextFormatter instance for the primary string, passing in an array that contains the formatter instance for the variable substring.

The following line from an English message catalog shows a string entry with two substitutions and entries to support the localizable variable at index 0 (the second variable in the string, the account number, does not need to be localized):

```
successfulTransaction = The {0} operation on account {1} was successful.
depositOpString = deposit
withdrawOpString = withdrawal
```

The following code shows the creation of the nested formatter instance and its insertion (with the account number variable) into the primary formatter instance:

```
public void updateAccount(String transactionType) {
   ...
   // Successful deposit
   LocalizableTextFormatter opLTF =
      new LocalizableTextFormatter("BankingResources",
                                   "depositOpString",
                "BankingSample");
   Object[] args = {opLTF, new String(this.accountNumber)};
   ...
   LocalizableTextFormatter successLTF =
      new LocalizableTextFormatter ("BankingResources",
                                    "successfulTransaction",
                                    "BankingSample",
                                    args);
   ...
   successLTF.format(this.applicationLocale);
   ...
}
```

Generating localized text

Create a formatter instance and set localization values as needed.

1. If needed, customize the formatting behavior.
2. In application code, call the appropriate format method.

You can provide fallback behavior for use if the appropriate message catalog is not available at formatting time.

The following code generates a localized string. If the formatting fails, the application retrieves and uses a fallback string instead of the localized string:

```
import com.ibm.websphere.i18n.localizabletext.LocalizableException;
import com.ibm.websphere.i18n.localizabletext.LocalizableTextFormatter;
import java.util.Locale;

public void drawAccountNumberGUI(String accountType){
   ...
   LocalizableTextFormatter ltf = new LocalizableTextFormatter();
   ...
   ltf.setFallBackString("Enter account number: ");
   try {
      msg = new Label(ltf.format(this.applicationLocale), Label.CENTER);
   }
   catch (LocalizableException le) {
      msg = new Label(ltf.getFallBackString(), Label.CENTER);
   }
   ...
}
```

When the application is finished, deploy your application.For more information, see "Preparing the localizable-text package for deployment."

Customizing the behavior of a formatting method

You can customize formatting behavior by passing your own formatter classes into a LocalizableTextFormatter instance through an array of optional values. This enables you to take variables other than locale and time zone into account when formatting localized text.

1. Write your own formatter class. For more information about implementation, see "LocalizableTextFormatter class."
2. In application code, create an instance of your formatter class as appropriate and pass it with any other optional localization values into an instance of LocalizableTextFormatter. When the LocalizableTextFormatter instance reads the instance that has been passed in, it attempts to call format() on the passed-in instance. The string returned is then processed with any other elements in the array.

The localizable-text package provides an example of a user-defined class, called LocalizableTextDateTimeArgument. This class enables date and time information to be selectively formatted according to the style values defined in java.text.DateFormat as well as constants defined within LocalizableTextDateTimeArgument itself.

Preparing the localizable-text package for deployment

Write code to compose the language-specific strings.

The LocalizableTextEJBDeploy tool is used to create a deployment JAR for the Localizable Text service. You must deploy the enterprise bean in each enterprise application that requires support for localized text.

1. Make sure the **LocalizableTextEJBDeploy** tool (ltext.jar) exists in the lib directory under the product's installation root directory.
2. Set up a working directory for the **LocalizableTextEJBDeploy** tool to use. You will need to pass this location to the tool through a command-line interface.
3. Run the **LocalizableTextEJBDeploy** tool. You might be asked if you want to regenerate deployment code for the LocalizableText bean. Do not redeploy the bean; if you do, the generated JNDI name will be incorrect.

   To deploy the bean on multiple hosts and servers, run the tool for each host/server combination. This generates a unique JNDI name for each deployment. After the tool is run, a deployment JAR is located in the working directory you specified.

5.0.2 + Using the Assembly Toolkit, assemble the deployment JAR in an enterprise application with other application components.

5.0.1 Using the Application Assembly Tool (AAT), assemble the deployment JAR in an enterprise application with other application components.

As part of preparing for deployment, verify the following:
- Add the resource bundles for your application to the EAR as files.
- Add the location of the EAR to the server's class path. This is so the resource bundles can be located on the virtual host and server.

The same deployment JAR can be included in several enterprise applications.

LocalizableTextEJBDeploy command

This topic describes the command-line syntax for the **LocalizableTextEJBDeploy** tool. The file that contains this tool (ltext.jar) must be located in the lib directory of the product installation root.

```
LocalizableTextEJBDeploy
   -a applicationName
   -h virtualHostName
   -i installationDirectory
   -s serverName
   -w workingDirectory
```

Parameters

The required parameters, which can be specified in any order, follow:

applicationName
: The name of the formatting session bean. This name is used in LocalizableTextFormatter instances to specify where the actual formatting takes place. If the name cannot be resolved at run time, the format method throws an exception.

virtualHostName
: The name of the virtual host on which the formatting session bean is deployed. This value is case-sensitive on all operating platforms.

installationDirectory
: The location at which the application server product is installed.

serverName
: The name of the application server. If this argument is not specified, the default server name for the product is used.

workingDirectory
: A location for the tool to use temporarily.

Internationalization: Resources for learning

Use the following links to find relevant supplemental information about internationalization. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to this product but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- Programming instructions and examples
- Programming specifications

Programming instructions and examples
- Java internationalization tutorial

  An online tutorial that explains how to use the Java 2 SDK Internationalization API.

Programming specifications
- Java 2 SDK, Standard Edition Documentation: Internationalization

  The Java internationalization documentation from Sun Microsystems, including a list of supported locales and encodings.
- Making the WWW truly World Wide

  The W3C's effort to make World Wide Web technology work with the many writing systems, languages, and cultural conventions of the global community:
- developerWorks - Unicode

  Articles on various subjects relating to Unicode, from IBM's developerWorks.

Chapter 13. Using the transaction service

These topics provide information about using transactions with WebSphere applications

WebSphere applications can use transactions to coordinate multiple updates to resources as atomic units (as indivisible units of work) such that all or none of the updates are made permanent.

In WebSphere Application Server, transactions are handled by three main components:

- A transaction manager that supports the enlistment of recoverable XAResources and ensures that each such resource is driven to a consistent outcome either at the end of a transaction or after a failure and restart of the application server. In addition, WebSphere Application Server for z/OS V5 supports the coordination of resource managers through RRS (z/OS resource recovery services).
- A container in which the J2EE application runs. The container manages the enlistment of XAResources on behalf of the application when the application performs updates to transactional resource managers (for example, databases). Optionally, the container can control the demarcation of transactions for enterprise beans configured for container-managed transactions.
- An application programming interface (UserTransaction) that is available to bean-managed enterprise beans and servlets. This allows such application components to control the demarcation of their own transactions.

For more information about using transactions with WebSphere applications, see the following topics:

- Transaction support in WebSphere Application Server
- Using local transactions
- Developing a WebSphere application to use transactions
- Classifying WebSphere transaction workload for WLM
- Configuring transaction properties for an application server
- Managing active transactions
-
- Interoperating transactionally between application servers
- Troubleshooting transactions
- Transaction service exceptions
- UserTransaction interface - methods available

Transaction support in WebSphere Application Server

A transaction is unit of activity within which multiple updates to resources can be made atomic (as an indivisible unit of work) such that all or none of the updates are made permanent. For example, multiple SQL statements to a relational database are committed atomically by the database during the processing of an SQL COMMIT statement. In this case, the transaction is contained entirely within the database manager and can be thought of as a resource manager local transaction (RMLT). In some contexts, a transaction is referred to as a logical unit of work (LUW).

The way that applications use transactions depends on the type of application component, as follows:

© Copyright IBM Corp. 2003

- A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) use bean-managed transactions.

WebSphere Application Server is a transaction manager that supports the coordination of resource managers through their XAResource interface and participates in distributed global transactions with other OTS 1.2 compliant transaction managers (for example J2EE 1.3 application servers). WebSphere applications can also be configured to interact with databases, JMS queues, and JCA connectors through their local transaction support when distributed transaction coordination is not required.

In addition to supporting the coordination of XAResource-based resource managers, WebSphere Application Server for z/OS V5 supports the coordination of resource managers through RRS (z/OS resource recovery services). RRS-compliant resource managers include DB2, MQSeries, IMS, and CICS. IBM WebSphere Application Server for z/OS is capable of coordinating a mix of RRSTransactional resource managers and XA capable resource managers under the same global transaction.

Resource managers that offer transaction support can be categorized into those that support two-phase coordination (by offering an XAResource interface or by supporting RRS) and those that support only one-phase coordination (for example through a LocalTransaction interface). The WebSphere Application Server transaction support provides coordination, within a transaction, for any number of two-phase capable resource managers. It also enables a single one-phase capable resource manager to be used within a transaction in the absence of any other resource managers, although a WebSphere transaction is not necessary in this case.

You can use transaction classes to classify client workload for workload management. The workload is different WebSphere transactions targeted to separate servant regions, each with goals defined by appropriate service classes. Each transaction is dispatched in its own WLM enclave in a servant region process, and is managed according to the goals of its service class. The server controller, which workload management views as a queue manager, uses the enclave associated with a client request to manage the priority of the work. If the work has a high priority, workload management can direct the work to a high-priority servant in the server. If the work has a low priority, workload management can direct the work to a low-priority servant. The effect is to partition the work according to priority within the same server.

Under normal circumstances you cannot mix one-phase commit capable resources and two-phase commit capable resources in the same global transaction, because one-phase commit resources cannot support the prepare phase of two-phase commit. There are some special circumstances where it is possible to include mixed-capability resources in the same global transaction:

- In scenarios where there is only a single one-phase commit resource provider that participates in the transaction and where all the two-phase commit resource-providers that participate in the transaction are used in a read-only fashion. In this case, the two-phase commit resources all vote read-only during the prepare phase of two-phase commit. Because the one-phase commit resource provider is the only provider to actually perform any updates, the one-phase commit resource does not need to be prepared.

- In scenarios where there is only a single one-phase commit resource provider that participates in the transaction with one of more two-phase commit resource providers and where last participant support is available.

  Last participant support (of WBI Server Foundation) enables the use of a single one-phase commit capable resource with any number of two-phase commit capable resources in the same global transaction.

Resource manager local transaction (RMLT)

A resource manager local transaction (RMLT) is a resource manager's view of a local transaction; that is, it represents a unit of recovery on a single connection that is managed by the resource manager.

Resource managers include:

- Enterprise Information Systems that are accessed through a resource adapter, as described in the J2EE Connector Architecture 1.0.
- Relational databases that are accessed through a JDBC datasource.
- JMS queue and topic destinations.

Resource managers offer specific interfaces to enable control of their RMLTs. J2EE connector resource adapters that include support for local transactions provide a LocalTransaction interface to enable applications to request that the resource adapter commit or rollback RMLTs. JDBC datasources provide a Connection interface for the same purpose.

The boundary at which all RMLTs must be complete is defined in WebSphere Application Server by a local transaction containment (LTC).

Global transactions

If an application uses two or more resources, then an external transaction manager is needed to coordinate the updates to both resource managers in a global tansaction.

Global transaction support is available to web and enterprise bean J2EE components. Enterprise bean components can be subdivided into beans that exploit container-managed transactions (CMT) or bean-managed transactions (BMT).

BMT enterprise beans and web components can use the Java Transaction API (JTA) UserTransaction interface to define the demarcation of a global transaction. The UserTransaction interface is obtained by a JNDI lookup of `java:comp/UserTransaction`. The UserTransaction is not available to the following components:

- CMT enterprise beans. Any attempt by such beans to obtain the interface results in an exception in accordance with the EJB specification.
- Client applications running outside the Web and EJB containers.

Ensure that programs that perform a JNDI lookup of the UserTransaction interface, use an InitialContext that resolves to a local implementation of the interface. Also ensure that such programs use a JNDI location appropriate for the EJB version.

Before the EJB 1.1 specification, the JNDI location of the UserTransaction interface was not specified. Each EJB container implementor defined it in an implementation-specific manner. Earlier versions of WebSphere Application Server, up to and including Version 3.5.x (without EJB 1.1), bind the UserTransaction interface to a JNDI location of jta/usertransaction. WebSphere Application Server Version 4, and later releases, bind the UserTransaction interface at the location defined by EJB 1.1, which is java:comp/UserTransaction. WebSphere Application

Server, Version 5 no longer provides the jta/usertransaction binding within Web and EJB containers to applications at a J2EE level of 1.3 or later. For example, EJB 2.0 applications can use only the java:comp/UserTransaction location.

Local transaction containment (LTC)

A local transaction containment (LTC) is used to define the application server behavior in an unspecified transaction context.

(Unspecified transaction context is defined in the Enterprise JavaBeans 2.0 Specification.)

A LTC is a bounded unit-of-work scope within which zero, one, or more resource manager local transactions (RMLTs) can be accessed. The LTC defines the boundary at which all RMLTs must be complete; any incomplete RMLTs are resolved, according to policy, by the container. An LTC is local to a bean instance; it is not shared across beans even if those beans are managed by the same container. LTCs are started by the container before dispatching a method on a J2EE component (such as an enterprise bean or servlet) whenever the dispatch occurs in the absence of a global transaction context. LTCs are completed by the container depending on the application-configured LTC boundary; for example at the end of the method dispatch. There is no programmatic interface to the LTC support; rather LTCs are managed exclusively by the container and configured by the application deployer through transaction attributes in the application deployment descriptor.

A local transaction containment cannot exist concurrently with a global transaction. If application component dispatch occurs in the absence of a global transaction, the container always establishes an LTC. The only exceptions to this behavior is when an application component dispatch occurs without container interposition; for example, for a stateless session bean `create`.

Local and global transaction considerations

Applications use resources, such as JDBC data sources or connection factories, that are configured through the Resources view of the WebSphere Application Server Administrative Console. How these resources participate in a global transaction depends on the underlying transaction support of the resource provider. For example, most JDBC providers can provide either XA or non-XA versions of a data source. A non-XA data source can support only resource manager local transactions (RMLTs), but an XA data source can support two-phase commit coordination, as well as local transactions.

Additionally, some JDBC Providers such as the DB2 for z/OS Local JDBC Provider support the use of z/OS Resource Recovery Service (RRS) to coordinate transaction processing. This type of JDBC Provider is RRSTransactional. When RRS is used, both local and global transactions are supported.

If an application uses two or more resource providers that support only RMLTs, then atomicity cannot be assured because of the one-phase nature of these resources. To ensure atomic behavior, the application should use resources that support XA coordination or RRS coordination and should access them within a global transaction.

If an application uses only one RMLT, the atomic behavior can be guaranteed by the resource manager, which can be accessed under a local transaction containment context.

An application can also access a single resource manager under a global transaction context, even if that resource manager does not support the XA coordination. An application can do this, because WebSphere Application Server performs an "only resource optimization" and interacts with the resource manager under a RMLT. Within a global transaction context, any attempt to use more than one resource provider that supports only RMLTs causes the global transaction to be rolled back.

At any moment, an instance of an enterprise bean can have work outstanding in either a global transaction context or a local transaction containment context, but never both. An instance of an enterprise bean can change from running under one type of context to the other (in either direction), if all outstanding work in the original context is complete. Any violation of this principle causes an exception to be thrown when the enterprise bean tries to start the new context.

Developing components to use transactions

These topics provide information about developing WebSphere application components to use transactions

The way that applications use transactions depends on the type of application component, as follows:
- A session bean can either use container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) use bean-managed transactions.

You configure whether a component uses container- or bean-managed transactions by setting an appropriate value on the Transaction type deployment attribute, as described in Configuring transactional deployment attributes using the Assembly ToolkitorConfiguring transactional deployment attributes using the Application Assembly Tool. You can also configure other transactional deployment descriptor attributes.

If you want a session bean to manage its own transactions, you must write the code that explicitly demarcates the boundaries of a transaction as described in Using bean-managed transactions.

Similarly, if you want a Web component to use transactions, you must write the code that explicitly demarcates the boundaries of a transaction as described in Using bean-managed transactions.

Configuring transactional deployment attributes using the Assembly Toolkit

Use this task to configure the transactional deployment descriptor attributes associated with an EJB or Web module, to enable a J2EE application to use transactions.

This topic describes the use of the Assembly Toolkit to configure the deployment attributes of an application. This task description assumes that you have an EAR file for an application component, that can be deployed in WebSphere Application Server. For more details about using the Assembly Toolkit, see Assembling applications with the Assembly Toolkit.

To set transactional attributes in the deployment descriptor for an application component (enterprise bean or servlet), complete the following steps:

1. Start the Assembly Toolkit.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, use the import wizard to import the EAR file into the Assembly Toolkit. To start the import wizard:
   a. Click **File-> Import-> EAR file**
   b. Click **Next**, then select the EAR file.
   c. Click **Finish**.
3. In the J2EE Hierarchy view, right-click the component instance, then click **Open With > Deployment Descriptor Editor**. For example:
   - For a session bean, expand **EJB Modules->** *ejb_module_instance***->** **Session Beans** then select the bean instance.
   - For a servlet, expand **Web Modules->** *web_application***->** *web component* then select the servlet instance.

   A property dialog notebook for the component is displayed in the property pane.
4. Set the **Transaction type** attribute, which defines the transactional manner in which the container invokes a method. You can set this attribute to Container or Bean, as follows:
   - For a session bean to use container-managed transactions, set Container
   - For a session bean to use bean-managed transactions, set Bean
   - For an entity bean, set Container
   - For a Web component (servlet), set Bean
5. In the property pane, select the IBM Extensions tab.
6. Under **WebSphere Extensions**, configure J2EE component extensions attributes for extended local transaction containment. To enable management of local transaction containments, configure the following EJB extensions attributes. These attributes configure, for the component, the behavior of the container's local transaction containment (LTC) environment that the container establishes whenever a global transaction is not present.

   Boundary
   : Specifies the duration of a local transaction context. You can set this attribute to either **Bean method**.

   Resolver
   : Specifies how the local transaction is to be resolved before the local transaction context ends: by the application through user code or by the EJB container. You can set this attribute to either **Application** or **ContainerAtBoundary**.

   Unresolved action
   : Specifies the action that the container must take when the local transaction context scope ends, if resources are uncommitted by an application in a local transaction and the **Resolution control** is set to Application. You can set this attribute to either **Commit** or **Rollback**.
7. [For EJB components only] For container-managed transactions, configure how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean's business method:
   a. In the navigation pane, click the Assembly Descriptor tab. The **Container Transactions** box displays a table of the methods for enterprise beans.
   b. For each method of the enterprise bean set the **Transaction attribute** attribute to an appropriate value.
8. Save your changes to the deployment descriptor.

a. Close the deployment descriptor editor.
b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.

9. Verify the archive files.
10. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
11. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard. Note that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Important: Use **Run On Server** for unit testing only. Assembly Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites the server configuration file for that server. Do not use on production servers.

    For instructions on remote testing, see the article "Setting Up a Remote WebSphere Application Server in WebSphere Studio V5" at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

After assembling your application, use a systems management tool to deploy the EAR file onto the application server that is to run the application; for example, using the administrative console as described in Deploying and managing applications.

Configuring transactional deployment attributes using the Application Assembly Tool

Use this task to configure the transactional deployment descriptor attributes associated with an EJB or Web module, to enable a J2EE application to use transactions.

To set transactional attributes in the deployment descriptor for an application component (enterprise bean or servlet), complete the following steps:

1. Start the Application Assembly Tool.
2. Create or edit the application EAR file. For example, to change attributes of an existing application, click **File-> Open** then select the EAR file.
3. In the navigation pane, select the component instance; for example:
    - For a session bean, expand **ejb_module_instance-> Session beans** then select the bean instance.
    - For a servlet, expand **web_application-> Web Components** then select the servlet instance.

    A property dialog notebook for the component is displayed in the property pane.
4. In the property pane, select the Advanced tab.
5. Set the **Transaction type** attribute, which defines the transactional manner in which the container invokes a method. You can set this attribute to Container or Bean, as follows:
    - For a session bean to use container-managed transactions, set Container
    - For a session bean to use bean-managed transactions, set Bean

- For an entity bean, set Container
- For a Web component (servlet), set Bean

6. In the property pane, select the IBM Extensions tab.
7. Configure J2EE component extensions attributes for extended local transaction containment. To enable management of local transaction containments, configure the following EJB extensions attributes. These attributes configure, for the component, the behavior of the container's local transaction containment (LTC) environment that the container establishes whenever a global transaction is not present.

   Boundary
   Specifies the duration of a local transaction context. You can set this attribute to either **Bean method**, as described in Entity bean assembly settings.

   Resolution control
   Specifies how the local transaction is to be resolved before the local transaction context ends: by the application through user code or by the EJB container. You can set this attribute to either **Application** or **ContainerAtBoundary**, as described in Entity bean assembly settings.

   Unresolved action
   Specifies the action that the container must take when the local transaction context scope ends, if resources are uncommitted by an application in a local transaction and the **Resolution control** is set to Application. You can set this attribute to either **Commit** or **Rollback**, as described in Entity bean assembly settings.
8. [For EJB components only] For container-managed transactions, configure how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean's business method:
   a. In the navigation pane, select **Container Transactions**. This displays a table of the methods for enterprise beans.
   b. For each method of the enterprise bean set the **Transaction attribute** attribute to an appropriate value, as defined in Container transaction assembly settings.

Using bean-managed transactions

This topic describes how to enable a session bean or servlet to use bean-managed transactions, to manage its own transactions directly instead of letting the container manage the transactions.

Note: Entity beans cannot manage transactions (so cannot use bean-managed transactions).

To enable a session bean or servlet to use bean-managed transactions, complete the following steps:

1. 5.0.2 + Set the **Transaction type** attribute in the component's deployment descriptor to `Bean`, as described in Setting transactional attributes in the deployment descriptor.
2. 5.0.1 Set the **Transaction type** attribute in the component's deployment descriptor to `Bean`, as described in Setting transactional attributes in the deployment descriptor.
3. Write the component code to actively manage transactions When writing the code required by a component to manage its own transactions, remember the following basic rules:

- An instance of a stateless session bean cannot reuse the same transaction context across multiple methods called by an EJB client.
- An instance of a stateful session bean can reuse the same transaction context across multiple methods called by an EJB client.

The following code extract shows the standard code required to obtain an object encapsulating the transaction context. There are three basics steps involved:
- The component class must set the value of the javax.ejb.SessionContext object reference in the setSessionContext method.
- A javax.transaction.UserTransaction object is created by calling a lookup on ″java:comp/UserTransaction″.
- The UserTransaction object is used to participate in the transaction by calling transaction methods such as begin and commit as needed. If an enterprise bean begins a transaction, it must also complete that transaction either by invoking the commit method or the rollback method.

Code example: Getting an object that encapsulates a transaction context

```
...
import javax.transaction.*;
...
public class MyStatelessSessionBean implements SessionBean {
private SessionContext mySessionCtx =null;
...
public void setSessionContext (SessionContext ctx)throws EJBException {
mySessionCtx =ctx;
}
...
    public float doSomething(long arg1)throws FinderException,EJBException {
        UserTransaction userTran = (UserTransaction)initCtx.lookup(
           "java:comp/UserTransaction");
        ...
        //User userTran object to call transaction methods
        userTran.begin ();
        //Do transactional work
        ...
        userTran.commit ();
        ...
    }
    ...
}
```

Classifying WebSphere transaction workload for WLM

This topic describes how to use transaction classes to classify client workload for workload management. The workload is different WebSphere transactions targeted to separate servant regions, each with goals defined by appropriate service classes. Each transaction is dispatched in its own WLM enclave in a servant region process, and is managed according to the goals of its service class.

This topic describes steps to classify transaction workload as a way of managing the workload service objectives. You also need to define the service objectives (goals) for the service classes used. In addition, you must define the service objectives of the WebSphere for z/OS servers and your business application servers.

For more information about defining service objectives (goals) for each service class, see the *z/OS MVS Planning: Workload Management* book, SA22-7602, for

example at http://publibz.boulder.ibm.com/epubs/pdf/iea2w131.pdf, or the z/OS WLM Web page at http://www.ibm.com/servers/eserver/zseries/zos/wlm/.

You can classify your WebSphere work using the WLM CB-type classification criteria:
- Server name (CN)
- Server instance name (SI)
- User ID assigned to the transaction (UI)
- Transaction class (TC)

Note: To get started, you do not need to define special classification rules and work qualifiers, but you may want to do this for your production system.

To classify work using server and userid criteria, you use a combination of the WLM Workload Classification rules in the WLM ISPF dialog panels. For more information about defining WLM Classification rules, see the section "Workload management and WebSphere for z/OS" in the *Installation and Customization Guide*, which includes an example of classification rules.

To classify work using transaction classes, you define and use transaction class mappings, as described in this task. The steps to classify work using transaction classes are:

1. Define transaction class mappings based on the HTTP virtual host name, port number, and URI (Universal Resource Identifier - encoded address for any resource on the Web) provided with each work HTTP or HTTPS request.
   a. Create a Transaction Class mapping file (as a simple text file). For example: `/wasconfig/t5was/MyTrMapFile.txt`
   b. Edit the Transaction Class mapping file to define each transaction class mapping that you want to use. Define each mapping on a separate line, using the following syntax:
   ```
   TransClassMap host:port uritemplate tclass
   ```
   Note: In the host or port fields, you can use wildcard characters only for the entire field as shown in the following example.

   This syntax is the same syntax as for WebSphere for z/OS Version 4.0.1. For more information about this syntax, see Transaction class mapping file entries. For example:
   ```
   TransClassMap wsc4.washington.ibm.com:9080  /MyIVT/index.*    TCLMYIVT
   TransClassMap wsc4.washington.ibm.com:9080  /MyIVT/ivtejb     TCLMYEJB
   TransClassMap wsc4.washington.ibm.com:*     /SuperSnoop*      TCLSNOOP
   TransClassMap wsc4.washington.ibm.com:*     /ssb/*            TCLSSB
   TransClassMap *:*                           /admin*           TCLADMIN
   ```
2. Specify the Transaction Class mapping file on the administrative properties for each server that is to handle work classified by transaction class. To specify the Transaction Class mapping file for a server, use the administrative console to complete the following steps:
   a. In the navigation pane, click **Servers > Application Servers**.
   b. In the content pane, select the server instance, *server_name*.
   c. In the Additional Properties list in the contents pane, select **Web Container**.
   d. In the Additional Properties list for the Web container, select **Advanced Settings**.

e. In the **Transaction Class Mapping** field, type the name of the Transaction Class mapping file that you edited in an earlier step. For example: `/wasconfig/t5was/MyTrMapFile.txt`

   This sets the following variable in the server's was.env file:

   `protocol_http_transport_class_mapping_file``=/wasconfig/t5was/MyTrMapFile.txt`

f. If you want to use a transaction class to classify outbound data that is delivered in response to HTTP and HTTPS requests, select the TCLASS option in the **Network QoS** field. If you specify TCLASS, WebSphere for z/OS uses the transaction class value that was used to classify the inbound request to the z/OS Workload Manager.

The following table shows classification rules for CB-type work in which the default service class is WSMED and has a reporting class of RWSDEFLT. Work run in the WSPROD WebSphere server is classified as WSMED with a reporting class of RWSPROD, unless it has a transaction class of TCLASS1, TCLASS2, or TCLASS2 assigned through the transaction class mapping file below.

```
Qualifier    Qualifier Start       Service  Report
# type       name      position    Class    Class
- ---------  --------  --------    -------- --------
                          Default: WSMED    RWSDEFLT
1 CN         WSPROD    1           WSMED    RWSPROD
2 . TC       . TCLASS1             WSFAST   RWSPRD1
2 . TC       . TCLASS2             WSMED    RWSPRD2
2 . TC       . TCLASS5             WSSLOW   RWSPRD5
1 CN         WSTEST    1           WSSLOW   RTSTEST
2 . UI       . USER1               WSMED    RTSTSTU2
2 . TC       . TCLASS5             WSSLOW   RTSTST5
```

The following table shows how work can be assigned a transaction class based on its host name, port number, or URI. For example, a web request of http://ibm.com:80/Webap1/myservlet handled by the WSPROD server would be assigned a transaction class of TCLASS1, a service class of WSFAST, and a reporting class of RWSPRD1 by the classification rules shown above.

```
TransClassMap www.ibm.com:80 /Webap1/myservlet TCLASS1
TransClassMap www.ibm.com:* /Webap1/myservlet TCLASS2
TransClassMap *:443 * TCLASS3
TransClassMap *:* /Webap1/myservlet TCLASS4
TransClassMap www.ibm.com:* /Webap5/* TCLASS5
TransClassMap * * TCLASS6
```

Configuring transaction properties for an application server

Use this task to configure the transaction properties for an application server; for example, to define the location of the directory that contains the transaction log or to change default timeouts associated with transactions.

To configure the transaction properties for an application server, complete the following steps:

1. Start the Administrative console
2. In the navigation pane, select **Servers-> Manage Application Servers->** ***your_app_server*** This displays the properties of the application server, *your_app_server*, in the content pane.
3. Select the Transaction Service tab, to display the properties page for the transaction service, as two notebook pages:

Configuration
: The values of properties defined in the configuration file. If you change these properties, the new values are applied when the application server next starts.

Runtime
: The runtime values of properties. If you change these properties, the new values are applied immediately, but are overwritten with the Configuration values when the application server next starts.

4. Select the Configuration tab, to display the transaction-related configuration properties.
5. In the **Total transaction lifetime timeout** field, type the number of seconds a transaction can remain inactive before it is ended by the transaction service. A value of 0 (zero) indicates that there is no timeout limit.
6. In the **Maximum transaction timeout** field, type the number of seconds for which a transaction started by or propagated into this application server may execute before it is ended by the transaction service. A value of 0 (zero) indicates that there is no timeout limit.
7. In the **Client inactivity timeout** field, type the number of seconds after which a client is considered inactive and the transaction service ends any transactions associated with that client. A value of 0 (zero) indicates that there is no timeout limit.
8. Click **OK**.
9. Stop then restart the application server.

   If you change the transaction log directory configuration property to an incorrect directory name, the application server will restart but be unable to open the transaction logs. You should change the configuration property to a valid directory name, then restart the application server.

Transaction service settings

Use this page to modify transaction service settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Transaction Service**.

Transaction log directory

Specifies the location of the JTA Partner Log.
This log is used for recovery of XA resources.

When the application that runs on the WebSphere product accesses XA resources, the WebSphere product stores information about the resource to enable XA transaction recovery.

Syntax
: [*location type URL tag*] *location specification*

where

- *location type URL tag* specifies the optional location type for the JTA Partner Log:
  - dir:// specifies that the JTA Partner Log location is in a fully qualified HFS directory specified by *location specification*. dir:// is the default.
- *location specification* specifies the location name for the JTA Partner Log:
  - If the *location type URL tag* is dir://, use a fully qualified HFS directory for the *location specification*. The complete name of the directory must be unique within the WebSphere node.

Default
: dir://*install root*/tranlog/*server name*

For additional information, see the *WebSphere Application Server for z/OS V5.0 Installation and Customization* manual.

Total transaction lifetime timeout

Specifies the maximum duration, in seconds, for container managed transactions started by this application server.
Component managed transactions that do not have a time-out explicitly set are also assigned this value.

Any transaction that is not requested to complete before this time-out is rolled back. If set to 0, only the maximum transaction timeout configuration value applies.

Data type	Integer
Units	Seconds
Default	120
Range	0 to 2 147 040

Client inactivity timeout

Specifies the maximum duration, in seconds, between transactional requests from a remote client.
Any period of client inactivity that exceeds this timeout results in the transaction rolling back in this application server. If set to 0, there is no timeout limit.

Data type	Integer
Units	Seconds
Default	60
Range	0 to 2 147 483 647

Enable logging for heuristic reporting

Select this property to enable the application server to log ″about to commit one-phase resource″ events from transactions that involve a one-phase commit resource and two-phase commit resources.
This property enables logging for heuristic reporting. If applications are configured to allow one-phase commit resources to participate in two-phase commit transactions, reporting of heuristic outcomes that occur at application server failure requires extra information to be written to the transaction log. If enabled, one additional log write is performed for any transaction that involves both one- and two-phase commit resources. No additional records are written for transactions that do not involve a one-phase commit resource.

Maximum Transaction Timeout

Specifies the maximum duration, in seconds, that transactions started by or propagated into this application server are allowed to execute.
This value limits the upper bound of all other transaction related time-outs. For example, assume a component attempts to set a transaction time-out of 360 seconds, and the Maximum Transaction Timeout setting is 300 seconds. The Maximum Transaction Time-out setting of 300 seconds is used.

Data type	Integer
Units	Seconds
Default	300
Range	0 to 2 147 040

Using local transactions

Local transaction containment (LTC) support, and its configuration through local transaction extended deployment descriptors, gives IBM WebSphere Application Server application programmers a number of advantages. This topic describes those advantages and how they relate to the settings of the local transaction extended deployment descriptors. This topic also describes points to consider to help you best configure transaction support for some example scenarios that use local transactions.

Develop an enterprise bean or servlet that accesses one or more databases that are independent and require no coordination.

If an enterprise bean does not need to use global transactions, it is often more efficient to deploy the bean with the Container Transaction deployment descriptor **Transaction** attribute set to `Not supported` instead of `Required`.

With the extended local transaction support of IBM WebSphere Application Server, applications can perform the same business logic in an unspecific transaction context as they can under a global transaction. An enterprise bean, for example, runs under an unspecified transaction context if it is deployed with a **Transaction** attribute of `Not supported` or `Never`.

The extended local transaction support provides a container-managed, implicit local transaction boundary within which application updates can be committed and their connections cleaned up by the container. Applications can then be designed with a greater degree of independence from deployment concerns. This makes using a **Transaction** attribute of `Supports` much simpler, for example, when the business logic may be called either with or without a global transaction context.

An application can follow a get-use-close pattern of connection usage regardless of whether or not the application runs under a transaction. The application can depend on the close behaving in the same way and not causing a rollback to occur on the connection if there is no global transaction.

There are many scenarios where ACID coordination of multiple resource managers is not needed. In such scenarios running business logic under a **Transaction** policy of `Not supported` performs better than if it had been run under a `Required` policy. This benefit is exploited through the **Local Transactions - Resolution-control** extended deployment setting of `ContainerAtBoundary`. With this setting, application interactions with resource providers (such as databases) are managed within implicit RMLTs that are both started and ended by the container. The RMLTs are committed by the container at the configured **Local Transactions - Boundary**; for example at the end of a method. If the application returns control to the container by an exception, the container rolls back any RMLTs that it has started.

This usage applies to both servlets and enterprise beans.

Use local transactions in a managed environment that guarantees clean-up.

Applications that want to control RMLTs, by starting and ending them explicitly, can use the default **Local Transactions - Resolution-control** extended deployment setting of `Application`. In this case, the container ensures connection cleanup at the boundary of the local transaction context.

J2EE specifications that describe application use of local transactions do so in the manner provided by the default setting of **Local Transactions -**

Resolution-control=Application and **Local Transactions - Unresolved-action**=Rollback. By configuring the **Local Transactions - Unresolved-action** extended deployment setting to Commit, then any RMLTs started by the application but not completed when the local transaction containment ends (for example, when the method ends) are committed by the container. This usage applies to both servlets and enterprise beans.

To determine how best to configure the transaction support for an application, depending on what you want to do with transactions, consider the following points.

General points

- You want to start and end global transactions explicitly in the application (BMT session beans and servlets only).

  For a session bean, set the **Transaction type** to Bean (to use bean-managed transactions) in the component's deployment descriptor. (You do not need to do this for servlets.)
- You want to access several XA resources atomically across one or more bean methods.

  In the Container transaction deployment descriptor, set **Transaction** to Required, Requires new, or Mandatory.
- You want to access several non-XA resources in a method and want to manage them independently.

  In the component's deployment descriptor, set **Local Transactions - Resolution-control** to Application and set **Local Transactions - Unresolved-action** to **Rollback**. In the Container transaction deployment descriptor, set **Transaction** to Not supported.

Points specific to **WBI Server Foundation**

- You want to use a single non-XA resource and one or more XAResources.

  Use the Last Participant Support of WBI Server Foundation.

Points specific to WebSphere Application Server for z/OS

- You want to use a non-XA resource along with multiple two-phase RRS resources.

  A non-XA resource in a transaction along with RRS resources is supported any time a global transaction is active. A global transaction is active when the deployment descriptor has **Transaction** set to Supports, Required, Requires New, or Mandatory. Global transactions also are active for Bean-Managed deployments.

Managing active transactions

Use this task to manage transactions that are active on an application server.

You can use this task to display a snapshot of all the transactions currently running on an application server. For each transaction, the following properties are shown: its local ID, global ID, and current status. The transaction status is shown as an integer value. The values correspond to the following status:

0 - active
1 - marked for rollback
2 - prepared
3 - committed
4 - rolled back
5 - unknown

6 - none
7 - preparing
8 - committing
9 - rolling back

You can also choose to finish transactions manually.

Under normal circumstances, transactions should run and complete (commit or rollback) automatically, without the need for intervention. However, in some circumstances, you may need to finish a transaction manually. For example, you may want to finish a transaction that has become stuck polling a resource manager that you know will not become available again within the desired timeframe.

Note: If you choose to finish a transaction on an application server, it is recorded as having completed in the transaction service logs for that server, so will not be eligible for recovery during server start up. If you finish a transaction, you are responsible for cleaning up any in-doubt transactions on the resource managers affected.

To manage the active transactions for an application server, use the administrative console to complete the following steps:

1. In the navigation pane, select **Servers-> Manage Application Servers** This displays a list of application servers in the content pane.
2. In the content pane, click *your_app_server* This displays the properties of the application server, *your_app_server*.
3. In the content pane, click the **Runtime** tab. This displays the runtime properties of the application server.
4. In the Additional Properties table, select **Transaction Service** This displays the runtime properties of the Transaction Service.
5. Click **Manage Transactions**. This displays a snapshot of all the transactions currently running on the server. For each transaction, the following properties are shown: its local ID, current status, and global ID.
6. If you want to finish one or more transactions, select the checkbox provided on the entry for the transaction, then click **Finish**. Alternatively, to finish all transactions, select the checkbox in the header of the transactions table, then click **Finish**.

Interoperating transactionally between application servers

This topic describes some considerations and actions that you can take to interoperate transactionally between different types of application servers.

To interoperate transactionally with a non-WebSphere application server, WebSphere Application Server switches dynamically between native transaction contexts and interoperable OTS contexts depending on the capability of the partner with which it is interoperating. WebSphere for z/OS always interoperates with other OTS contexts.

Troubleshooting transactions

Use this overview task to help resolve a problem that you think is related to the Transaction service.

To identify and resolve transaction-related problems, you can use the standard WebSphere Application Server RAS facilities. If you encounter a problem that you think might be related to transactions, complete the following steps:

1. Check for transaction messages in the administrative console. The Transaction service produces diagnostic messages prefixed by “WTRN”. The error message indicates the nature of the problem and provides some detail. The associated message information provides an explanation and any user actions to resolve the problem.
2. Check for Transaction messages in the activity log. Activity log messages produced by the Transaction service are accompanied by Log Analyzer descriptions.
3. Check for more messages in other potential message output repositories. For more information about a problem, check the standard output file configured by your administrator. This will contain more error messages and other detailed information about the problem.
4. Check for messages related to the application server’s transaction log directory when the problem occurred.

   Note: If you changed the transaction log directory and a problem caused the application server to fail (with in-flight transactions) before the server was restarted properly, the server will next start with the new log directory and be unable to automatically resolve in-flight transactions that were recorded in the old log directory. To resolve this, you can copy the transaction logs to the new directory then stop and restart the application server.
5. Check the RRS logs for any transaction activity involving RRS-compliant resources. IBM WebSphere Application Server for z/OS is capable of supporting both XA and RRS resource managers, and of coordinating a mix of RRSTransactional resource managers and XA capable resource managers under the same global transaction. If your installation uses XA resource managers, RRS resource managers, or a mixture of both, you can use the administrative console to view transaction logs that contain information about all transactions. You can find additional information about RRS transactions by using the RRS panels. For additional information, see:
   - *WebSphere Application Server for z/OS V5.0: Diagnosis*, GA22-7914, for information about diagnosing problems related to transactions.
   - *WebSphere Application Server for z/OS V5.0: Operations and Administration*, GA22-7912, for information about using the RRS panels and transaction logs.
   - *WebSphere Application Server for z/OS V5.0: Installation and Customization*, GA22-7910, for information about working with RRS logs and XA partner logs during restart and recovery mode.

Transaction service exceptions

This topic lists the exceptions that can be thrown by the WebSphere Application Server transaction service. The exceptions are listed in the following groups:

- Standard exceptions
- Heuristic exceptions

If the EJB container catches a system exception from the business method of an enterprise bean, and the method is running within a container-managed transaction, the container rolls back the transaction before passing the exception on to the client. For more information about how the container handles the exceptions thrown by the business methods for beans with container-managed transaction

demarcation, see the section *Exception handling* in the Enterprise JavaBeans 2.0 specification. That section specifies the container's action as a function of the condition under which the business method executes and the exception thrown by the business method. It also illustrates the exception that the client receives and how the client can recover from the exception.

Standard exceptions

The standard exceptions such as TransactionRequiredException, TransactionRolledbackException, and InvalidTransactionException are defined in the Java Transaction API (JTA) 1.0.1 Specification.

InvalidTransactionException
: This exception indicates that the request carried an invalid transaction context.

TransactionRequiredException exception
: This exception indicates that a request carried a null transaction context, but the target object requires an active transaction.

TransactionRolledbackException exception
: This exception indicates that the transaction associated with processing of the request has been rolled back, or marked for roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

Heuristic exceptions

A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service. Heuristic decisions are an issue only after the participant has been prepared and the second phase of commit processing is underway. Heuristic decisions are normally made only in unusual circumstances, such as repeated failures by the transaction manager to communicate with a resource manage during two-phase commit. If a heuristic decision is taken, there is a risk that the decision differs from the consensus outcome, resulting in a loss of data integrity.

The following list provides a summary of the heuristic exceptions. For more detail, see the Java Transaction API (JTA) 1.0.1 Specification.

HeuristicRollback exception
: This exception is raised on the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

HeuristicMixed exception
: This exception is raised on the commit operation to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

UserTransaction interface - methods available

For details about the methods available with the UserTransaction interface, see the WebSphere Application Server application programming interface reference information (Javadoc) or the Java Transaction API (JTA) 1.0.1 Specification.

Chapter 14. Using naming

Naming is used by clients of WebSphere Application Server applications most commonly to obtain references to objects related to those applications, such as Enterprise JavaBeans (EJB) homes. The following steps outline the context of Naming in the overall application development and deployment process. Steps for this task follow:

1. Develop your application using either JNDI or CosNaming (CORBA) interfaces.

   Use these interfaces to look up server application objects that are bound into the name space and obtain references to them. Most Java developers use the JNDI interface. However, the CORBA CosNaming interface is also available for performing Naming operations on WebSphere Application Server name servers or other CosNaming name servers.
2. Assemble your application using theAssembly ToolkitorApplication Assembly Tool (AAT). Application assembly is a packaging and configuration step that is a prerequisite to application deployment. If the application you are assembling is a client to an application running in another process, you should qualify the jndiName values in the deployment descriptors for the objects related to the other application. Otherwise, you may need to override the names with qualified names during application deployment. If the objects have fixed qualified names configured for them, you should use them so that the jndiName values do not depend on the other application's location within the topology of the cell.
3. Deploy your application

   Put your assembled application onto the application server. If the application you are assembling is a client to an application running in another server process, be sure to qualify the jndiName values for the other application's server objects if they are not already qualified.

   For more information on qualified names, see ″Lookup names support in deployment descriptors and thin clients.″
4. Configure name space bindings. This step is necessary in these cases:
   - Your deployed application is to be accessed by legacy client applications running on previous versions of WebSphere Application Server. In this case, you must configure additional name bindings for application objects relative to the default initial context for legacy clients. (Version 5 clients have a different initial context from legacy clients.)
   - The application requires qualified name bindings for such reasons as:
     – It will be accessed by J2EE client applications or server applications running in another server process.
     – It will be accessed by thin client applications.

     In this case, you can configure name bindings as additional bindings for application objects. The qualified names for the configured bindings are *fixed*, meaning they do not contain elements of the cell topology that can change if the application is moved to another server. Objects as bound into the name space by the system can always be qualified with a topology-based name. You must explicitly configure a name binding to use as a fixed qualified name.

© Copyright IBM Corp. 2003

For more information on qualified names, see ″Lookup names support in deployment descriptors and thin clients.″ For more information on configured name bindings, see ″Configured name bindings.″

5. Troubleshoot any problems that develop.

   If a Naming operation is failing and you need to verify whether certain name bindings exist, use the dumpNameSpace tool to generate a dump of the name space.

Naming

Naming is used by clients of WebSphere Application Server applications to obtain references to objects related to those applications, such as Enterprise JavaBeans (EJB) homes.

These objects are bound into a mostly hierarchical structure, referred to as a *name space*. In this structure, all non-leaf objects are called *contexts*. Leaf objects can be contexts and other types of objects. Naming operations, such as lookups and binds, are performed on contexts. All naming operations begin with obtaining an *initial context*. You can view the initial context as a starting point in the name space.

The name space structure consists of a set of *name bindings*, each consisting of a name relative to a specific context and the object bound with that name. For example, the name `myApp/myEJB` consists of one non-leaf binding with the name `myApp`, which is a context. The name also includes one leaf binding with the name `myEJB`, relative to `myApp`. The object bound with the name `myEJB` in this example happens to be an EJB home reference. The whole name `myApp/myEJB` is relative to the initial context, which you can view as a starting place when performing naming operations.

You can access and manipulate the name space through a *name server*. Users of a name server are referred to as *naming clients*. Naming clients typically use the Java Naming and Directory Interface (JNDI) to perform naming operations. Naming clients can also use the Common Object Request Broker Architecture (CORBA) CosNaming interface.

Typically, objects bound to the name space are resources and objects associated with installed applications. These objects are bound by the system, and client applications perform lookup operations to obtain references to them. Occasionally, server and client applications bind objects to the name space. An application can bind objects to transient or persistent partitions, depending on requirements.

In J2EE environments, some JNDI operations are performed with java: URL names. Names bound under these names are bound to a completely different name space which is local to the calling process. However, some lookups on the `java:` name space may trigger indirect lookups to the name server.

Version 5 features for name space support

The following are features of the WebSphere Application Server new to naming implementation as of Version 5:

- **Name space is distributed.**

  For additional scalability, the name space for a cell is distributed among various servers. Every server has a name server. In previous releases, there was only one name server for an entire administrative domain.

In WebSphere Application Server versions prior to V5, all servers shared the same default initial context, and everything was bound relative to that same initial context. In WebSphere Application Server V5, the default initial context for a server is its server root. System artifacts, such as EJB homes and resources, are bound to the server root of the server with which they are associated.

- **Transient and persistent partitions.**

  The name space is partitioned into transient areas and persistent areas. Server roots are transient. System-bound artifacts such as EJB homes and resources are bound under server roots. There is a cell persistent root, which you can use for cell-scoped persistent bindings, and a node persistent root, which you can use to bind objects with a node scope.
- **System name space structure.**

  The name space for the entire cell is federated among all servers in the cell. Every server process contains a name server. All name servers provide the same logical view of the cell name space. The various server roots and persistent partitions of the name space are interconnected by means of a system name space. You can use the system name space structure to traverse to any context in the cell name space.
- **Configured bindings.**

  You can use the configuration graphical interface and script interfaces to configure bindings in various root contexts within the name space. These bindings are read-only and are bound by the system at server startup.
- **Support for CORBA Interoperable Naming Service (INS) object URLs.**

  WebSphere Application Server V5 contains support for Common Object Request Broker Architecture (CORBA) object URLs (corbaloc and corbname) as Java Naming and Directory Interface (JNDI) provider URLs and lookup names.

Name space logical view

The name space for the entire cell is federated among all servers in the cell. Every server process contains a name server. All name servers provide the same logical view of the cell name space. The various server roots and persistent partitions of the name space are interconnected by a system name space. You can use the system name space structure to traverse to any context in a the cell's name space. A logical view of the name space is shown in the following diagram.

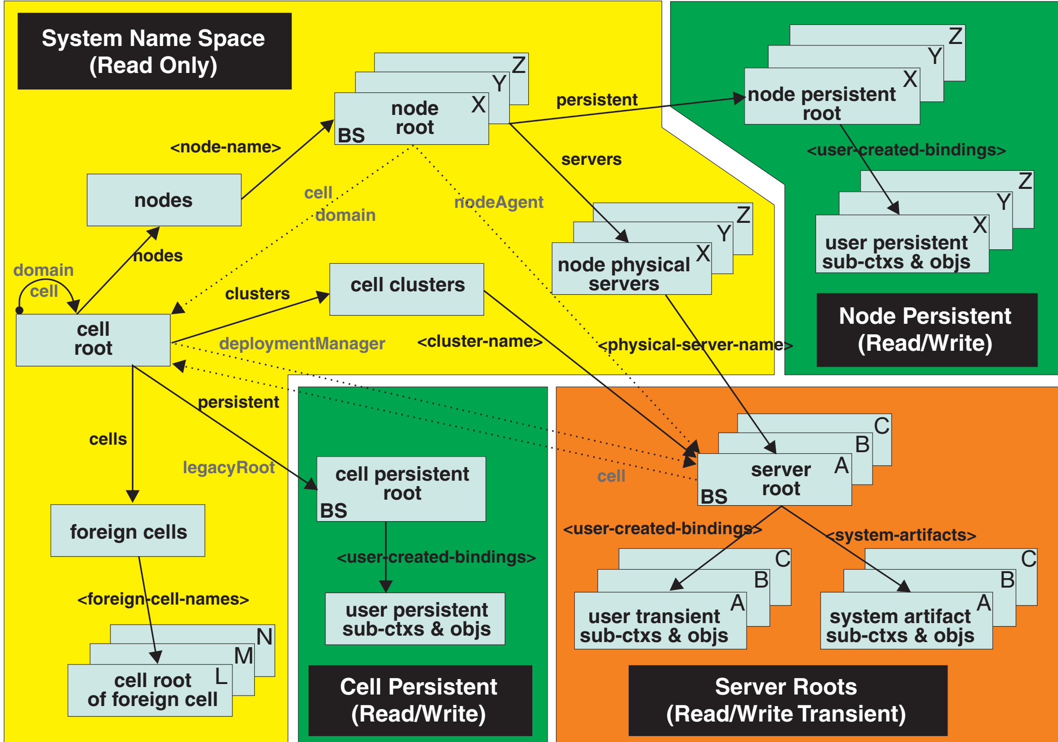


Figure 15. Name Space Logical View

The bindings in the preceding diagram appear with solid arrows, labeled in bold, and dashed arrows, labeled in gray. Solid arrows represent *primary bindings*. A primary binding is formed when the associated subcontext is created. Dashed arrows show *linked bindings*. A linked binding is formed when an existing context is bound under an additional name. Linked bindings are added for convenience or interoperability with previous WebSphere Application Server versions.

A cell name space is composed of contexts which reside in servers throughout the cell. All name servers in the cell provide the same logical view of the cell name space. A name server constructs this view at startup by reading configuration information. Each name server has its own local in-memory copy of the name space and does not require another running server to function. There are, however, a few exceptions. Server roots for other servers are not replicated among all the servers. The respective server for a server root must be running to access that server root context.

In WebSphere Application Server Network Deployment cells, the cell and node persistent areas can be read even if the deployment manager and respective node agent are not running. However, the deployment manager must be running to update the cell persistent segment, and a node agent must be running to update its respective node persistent segment.

Name space partitions

There are four major partitions in a cell name space:
- System name space partition
- Server roots partition
- Cell persistent partition

- Node persistent partition

System name space partition
: The system name space contains a structure of contexts based on the cell topology. The system structure supports traversal to all parts of a cell name space and to the cell root of other cells, which are configured as foreign cells. The root of this structure is the cell root. In addition to the cell root, the system structure contains a node root for each node in the cell. You can access other contexts of interest specific to a node from the node root, such as the node persistent root and server roots for servers configured in that node.

  All contexts in the system name space are read-only. You cannot add, update, or remove any bindings.

Server roots partition
: Each server in a cell has a server root context. A server root is specific to a particular server. You can view the server roots for all servers in a cell as being in a transient read/write partition of the cell name space. System artifacts, such as EJB homes for server applications and resources, are bound under the server root context of the associated server. A server application can also add bindings under its server root. These bindings are transient. Therefore, the server application creates all required bindings at application startup, so they exist anytime the application is running.

  A server cluster is composed of many servers that are logically equivalent. Each member of the cluster has its own server root. These server roots are not replicated across the cluster. In other words, adding a binding to the server root of one member does not propagate it to the server roots of the other cluster members. To maintain the same view across the cluster, you should create all user bindings under the server root by the server application at application startup so that the bindings are present under the server root of each cluster member. Because of Workload Management (WLM) behavior, a JNDI client outside a cluster has no control over which cluster member's server root context becomes the target of the JNDI operation. Therefore, you should execute bind operations to the server root of a cluster member from within that cluster member process only.

  Distributing application objects among many server roots is a departure from previous WebSphere Application Server releases, where all system artifacts were bound under a single root. This change can affect the names that clients use to look up these objects.

  Server-scoped bindings are relative to a server's server root.

Cell persistent partition
: The root context of the cell persistent partition is the cell persistent root. A binding created under the cell persistent root is saved as part of the cell configuration and continues to exist until it is explicitly removed. Applications that need to create additional persistent bindings of objects generally associated with the cell can bind these objects under the cell persistent root.

  It is important to note that the cell persistent area is not designed for transient, rapidly changing bindings. The bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

Note: In WebSphere Application Server Network Deployment cells, to bind objects to the cell persistent root, ensure that the deployment manager and all node agents in the cell are running.

An important role of the cell persistent root is as the initial context for clients running in previous WebSphere Application Server versions. If you want to access an enterprise bean by WebSphere Application Server v4.0.x and 3.5.x clients, you must ensure that a binding for it has been added to the cell persistent root. You can configure these additional bindings as cell-scoped bindings.

Node persistent partition

The node persistent partition is similar to the cell partition except that each node has its own node persistent root. A binding created under a node persistent root is saved as part of that node configuration and continues to exist until it is explicitly removed.

Applications that need to create additional persistent bindings of objects associated with a specific node can bind those objects under that particular node's node persistent root. As with the cell persistent area, it is important to note that the node persistent area is not designed for transient, rapidly changing bindings. These bindings are more static in nature, such as part of an application setup or configuration, and are not created at run time.

Note: In WebSphere Application Server Network Deployment cells, to bind objects to a node persistent root, ensure the node agent for the node is running.

Unlike the cell persistent root, the node persistent root plays no special role in interoperability with WebSphere Application Server clients of previous releases. Node-scoped bindings are relative to a node's node persistent root.

Note: In the system name space, there is no persistent node root for the deployment manager node because no node agent or application servers run in that node.

Initial context support

All naming operations begin with obtaining an initial context. You can view the initial context as a starting point in the name space. Use the initial context to perform naming operations, such as looking up and binding objects in the name space.

Initial contexts registered with the ORB as initial references

The server root, cell persistent root, cell root, and node root are registered with the name server's ORB and can be used as an initial context. An initial context is used by CORBA and enterprise bean applications as a starting point for name space lookups. The keys for these roots as recognized by the ORB are shown in the following table:

Root Context	Initial Reference Key
Server Root	NameServiceServerRoot
Cell Persistent Root	NameServiceCellPersistentRoot
Cell Root	NameServiceCellRoot, NameService

Node Root	NameServiceNodeRoot

A server root initial context is the server root context for the specific server you are accessing. Similarly, a node root initial context is the node root for the server being accessed.

You can use the previously mentioned keys in CORBA INS object URLs (corbaloc and corbaname) and as an argument to an ORB resolve_initial_references call. For examples, see CORBA and JNDI programming examples, which show how to get an initial context.

Default initial contexts

The default initial context depends on the type of client. Different categories of clients and the corresponding default initial context follow.

- **WebSphere Application Server V5 JNDI interface implementation**

  The JNDI interface is used by EJB applications to perform name space lookups. WebSphere Application Server clients by default use the WebSphere Application Server CosNaming JNDI plug-in implementation. The default initial context for clients of this type is the server root of the server specified by the provider URL. For more details, refer to the JNDI programming examples on getting initial contexts.
- **WebSphere Application Server JNDI interface implementation prior to V5**

  WebSphere Application Server clients running in releases prior to WebSphere Application Server V5 by default use WebSphere Application Server's v4.0 CosNaming JNDI plug-in implementation. The default initial context for clients of this type is the cell persistent root, also known as the *legacy root*.
- **Other JNDI implementation**

  Some applications can perform name space lookups with a non-WebSphere Application Server CosNaming JNDI plug-in implementation. Assuming the key **`NamingContext`** is used to obtain the initial context, the default initial context for clients of this type is the cell root.
- **CORBA**

  The standard CORBA client obtains an initial org.omg.CosNaming.NamingContext reference with the key **`NamingContext`**. The initial context in this case is the cell root.

Lookup names support in deployment descriptors and thin clients

Server objects, such as EJB homes, are bound relative to the server root context for the server in which the application is installed. Other objects, such as resources, can also be bound to a specific server root. The names used to look up these objects must be qualified so as to select the correct server root. This is a departure from previous versions of WebSphere Application Server, where these objects were all bound under a single root context. This section discusses what relative and qualified names are, when they can be used, and how you can construct them.

Relative names

All names are relative to a context. Therefore, a name that can be resolved from one context in the name space cannot necessarily be resolved from another context in the name space. This point is significant because the system binds objects with names relative to the server root context of the server in which the application is

installed. Each server has its own server root context. The initial JNDI context is by default the server root context for the server identified by the provider URL used to obtain the initial context. (Typically, the URL consists of a host and port.) For applications running in a server process, the default initial JNDI context is the server root for that server. A relative name will resolve successfully when the initial context is obtained from the server which contains the target object, but it will not resolve successfully from an initial context obtained from another server.

If all clients of a server application run in the same server process as the application, all objects associated with that application are bound to the same initial context as the clients' initial context. In this case, only names relative to the server's server root context are required to access these server objects. Frequently, however, a server application has clients that run outside the application's server process. The initial context for these clients can be different from the server application's initial context, and lookups on the relative names for server objects may fail. These clients need to use the qualified name for the server objects. This point must be considered when setting up the jndiName values in a J2EE client application deployment descriptors and when constructing lookup names in thin clients. Qualified names resolve successfully from any initial context in the cell.

Qualified names

All names are relative to a context. Here, the term *qualified name* refers to names that can be resolved from any initial context in a cell. This action is accomplished by using names that navigate to the same context, the cell root. The rest of the qualified name is then relative to the cell root and uniquely identifies an object throughout the cell. All initial contexts in a server (that is, all naming contexts in a server registered with the ORB as an initial reference) contain a binding with the name **cell**, which links back to the cell root context. All qualified names begin with the string **cell/** to navigate from the current initial context back to the cell root context.

A qualified name for an object is the same throughout the cell. The name can be topology-based, or some fixed name bound under the cell persistent root. Topology-based names, described in more detail below, navigate through the system name space to reach the target object. A fixed name bound under the cell persistent root has the same qualified name throughout the cell and is independent of the topology. Creating a fixed name under the cell persistent root for a server application object requires an extra step when the server application is installed, but this step eliminates impacts to clients when the application is moved to a different location in the cell topology. The process for creating a fixed name is described later in this section.

Generally speaking, you **must** use qualified names for EJB jndiName values in a J2EE client application deployment descriptors and for EJB lookup names in thin clients. The only exception is when the initial context is obtained from the server in which the target object resides. For example, a session bean which is a client to an entity bean can use a relative name if the two beans run in the same server. If the session bean and entity beans run in different servers, the jndiName for the entity bean must be qualified in the session bean's deployment descriptors. The same requirement may be true for resources as well, depending on the scope of the resource.

- **Topology-based names**

  The system name space partition in a cell's name space reflects the cell's topology. This structure can be navigated to reach any object bound into the

cell's name space. Topology-based qualified names include elements from the topology which reflect the object's location within the cell. For a system-bound object, such as an EJB home, the form for a topology-based qualified name depends on whether the object is bound to a single server or cluster. Both forms are described below.

Single Server
An object bound in a single server has a topology-based qualified name of the following form:

```
cell/nodes/nodeName/servers/serverName/relativeJndiName
```

where *nodeName* and *serverName* are the node name and server name for the server where the object is bound, and *relativeJndiName* is the unqualified name of the object; that is, the object's name relative to its server's server root context.

Server Cluster
An object bound in a server cluster has a topology-based qualified name of the following form:

```
cell/clusters/clusterName/relativeJndiName
```

where *clusterName* is the name of the server cluster where the object is bound, and *relativeJndiName* is the unqualified name of the object; that is, the object's name relative to a cluster member's server root context.

- **Fixed names**

  It is possible to create a fixed name for a server object so that the qualified name is independent of the cell topology. This quality is desirable when clients of the application run in other server processes or as pure clients. Fixed names have the advantage of not changing if the object is moved to another server. The jndiName values in deployment descriptors for a J2EE client application can reference the qualified fixed name for a server object regardless of the cell topology on which the client or server application is being installed.

  Defining a cell-wide fixed name for a server application object requires an extra step after the server application is installed. That is, a binding for the object must be created under the cell persistent root. A fixed name bound under the cell persistent root can be any name, but all names under the cell persistent root must be unique within the cell because the cell persistent root is global to the entire cell.

  A qualified fixed name has the form:

  ```
  cell/persistent/fixedName
  ```

  where *fixedName* is an arbitrary fixed name.

  The binding can be created programmatically (for example, using JNDI). However, it is probably more convenient to configure a cell-scoped binding for the server object.

  You must keep the programmatic or configured binding up-to-date. Configured EJB bindings are based on the location of the enterprise bean within the cell topology, and moving the EJB application to another single server or to a server cluster, for example, requires the configured binding to be updated. Similar changes affect an EJB home reference programmatically bound so that the fixed name would need to be rebound with a current reference. However, for J2EE clients, the jndiName value for the object, and for thin clients, the lookup name

for the object, remains the same. In other words, clients that access objects by fixed names are not affected by changes to the configuration of server applications they access.

JNDI support in WebSphere Application Server

IBM WebSphere Application Server includes a name server to provide shared access to Java components, and an implementation of the `javax.naming` JNDI package which supports user access to the WebSphere Application Server name server through the JNDI naming interface.

WebSphere Application Server does ***not*** provide implementations for:
- `javax.naming.directory` or
- `javax.naming.ldap` packages

Also, WebSphere Application Server does ***not*** support interfaces defined in the `javax.naming.event` package.

However, to provide access to LDAP servers, the development kit shipped with WebSphere Application Server supports Sun's implementation of:
- `javax.naming.ldap` and
- `com.sun.jndi.ldap.LdapCtxFactory`

WebSphere Application Server's JNDI implementation is based on version 1.2 of the JNDI interface, and was tested with Version 1.2.1 of Sun's JNDI Service Provider Interface (SPI).

The default behavior of this JNDI implementation is adequate for most users. However, users with specific requirements can control certain aspects of JNDI behavior.

Developing applications that use JNDI

References to EJB homes and other artifacts such as data sources are bound to the WebSphere name space. These objects can be obtained through the JNDI interface. Before you can perform any JNDI operations, you need to get an initial context. You can use the initial context to look up objects bound to the WebSphere name space.

These examples describe how to get an initial context and how to perform lookup operations.
- Getting the default initial context
- Getting an initial context by setting the provider URL property
- Setting the provider URL property to select a different root context as the initial context
- Looking up an EJB home with JNDI
- Looking up a JavaMail session with JNDI

In these examples, the default behavior of features specific to WebSphere's JNDI Context implementation is used.

WebSphere Application Server's JNDI context implementation includes special features. JNDI caching enhances performance of repeated lookup operations on the same objects. Name syntax options offer a choice of a name syntaxes, one optimized for typical JNDI clients, and one optimized for interoperability with

CosNaming applications. Most of the time, the default behavior of these features is the preferred behavior. However, sometimes you should modify the behavior for specific situations.

JNDI caching and name syntax options are associated with a javax.naming.InitialContext instance. To select options for these features, set properties that are recognized by the WebSphere Application Server's initial context factory. To set JNDI caching or name syntax properties which will be visible to WebSphere Application Server's initial context factory, follow the following steps.

1. Optional: Configure JNDI caches

   JNDI caching can greatly increase performance of JNDI lookup operations. By default, JNDI caching is enabled. In most situations, this default is the desired behavior. However, in specific situations, use the other JNDI cache options.

   Objects are cached locally as they are looked up. Subsequent lookups on cached objects are resolved locally. However, cache contents can become stale. This situation is not usually a problem, since most objects you look up do not change frequently. If you need to look up objects which change relatively frequently, change your JNDI cache options.

   JNDI clients can use several properties to control cache behavior.

   You can set properties:

   - From the command line by entering the actual string value. For example:

     ```
     java -Dcom.ibm.websphere.naming.jndicache.maxentrylife=1440
     ```

   - In a `jndi.properties` file by creating a file named `jndi.properties` as a text file with the desired properties settings. For example:

     ```
     ...
     com.ibm.websphere.naming.jndicache.cacheobject=none
     ...
     ```

     Include the file as the beginning of the classpath, so that the class loader loads your copy of `jndi.properties` before any other copies.

   - Within a Java program by using the **PROPS.JNDI_CACHE*** Java constants, defined in the ***com.ibm.websphere.naming.PROPS*** file. The constant definitions follow:

     ```
     public static final String JNDI_CACHE_OBJECT =
      "com.ibm.websphere.naming.jndicache.cacheobject";
     public static final String JNDI_CACHE_OBJECT_NONE      = "none";
     public static final String JNDI_CACHE_OBJECT_POPULATED = "populated";
     public static final String JNDI_CACHE_OBJECT_CLEARED   = "cleared";
     public static final String JNDI_CACHE_OBJECT_DEFAULT   =
      JNDI_CACHE_OBJECT_POPULATED;

     public static final String JNDI_CACHE_NAME =
      "com.ibm.websphere.naming.jndicache.cachename";
     public static final String JNDI_CACHE_NAME_DEFAULT = "providerURL";

     public static final String JNDI_CACHE_MAX_LIFE =
      "com.ibm.websphere.naming.jndicache.maxcachelife";
     public static final int    JNDI_CACHE_MAX_LIFE_DEFAULT = 0;

     public static final String JNDI_CACHE_MAX_ENTRY_LIFE =
      "com.ibm.websphere.naming.jndicache.maxentrylife";
     public static final int    JNDI_CACHE_MAX_ENTRY_LIFE_DEFAULT = 0;
     ```

     To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

```
java.util.Hashtable env = new java.util.Hashtable();
...
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
 // Disable caching
...
javax.naming.Context initialContext = new javax.naming.InitialContext(env);
```

2. Optional: Specify the name syntax

   Most WebSphere applications use JNDI to look up EJB objects and do not need to look up objects bound by CORBA applications. Therefore, the default name syntax used for JNDI names is the most convenient. If your application needs to look up objects bound by CORBA applications, you may need to change your name syntax so that all CORBA CosNaming names can be represented.

   JNDI clients can set the name syntax by setting a property. The property setting is applied by the initial context factory when you instantiate a new java.naming.InitialContext object. Names specified in JNDI operations on the initial context are parsed according to the specified name syntax.

   You can set the property:

   - From the command line by entering the actual string value. For example:

     ```
     java -Dcom.ibm.websphere.naming.name.syntax=ins
     ```

   - In a jndi.properties file by creating a file named jndi.properties as a text file with the desired properties settings. For example:

     ```
     ...
     com.ibm.websphere.naming.name.syntax=ins
     ...
     ```

     Include the file as the beginning of the classpath, so that the class loader loads your copy of jndi.properties before any other copies.

   - Within a Java program by using the PROPS.NAME_SYNTAX* Java constants, defined in the com.ibm.websphere.naming.PROPS file. The constant definitions follow:

     ```
     public static final String NAME_SYNTAX =
      "com.ibm.websphere.naming.name.syntax";
     public static final String NAME_SYNTAX_JNDI = "jndi";
     public static final String NAME_SYNTAX_INS  = "ins";
     ```

     To use the previous properties in a Java program, add the property setting to a hashtable and pass it to the InitialContext constructor as follows:

     ```
     java.util.Hashtable env = new java.util.Hashtable();
     ...
     env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS); // Set name syntax to INS
     ...
     javax.naming.Context initialContext = new javax.naming.InitialContext(env);
     ```

Example: Getting the default initial context

This example below gets the default initial context. That is, no provider URL is passed to the javax.naming.InitialContext constructor. The following section explains the process of determining the address of the bootstrap server to use to obtain the initial context.

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
...
Context initialContext = new InitialContext();
...
```

The default initial context returned depends the runtime environment of the JNDI client. The initial context returned in the various environments are listed below:

- Thin client: The server root context of the server running on the local host at port 2809.
- Pure client:
  - The context specified by the java.naming.provider.url property passed to launchClient command with the -CCD command line parameter. The context usually will be the server root context of the server at the address specified in the URL, although it is possible to construct a corbaname or corbaloc URL which resolves to some other context.
  - If no provider URL was specified, the server root context of the server running on the host and port specified by the -CCBootstrapHost -CCBootstrapPort command line parameters. The default host is the local host, and the default port is 2809.
- Server process: The server root context for that process.

Even though no provider URL is explicitly specified in the above example, the InitialContext may find a provider URL defined in other places that it searches for property settings.

Users of properties which affect ORB initialization should read the rest of this section for a deeper understanding of exactly how initial contexts are obtained, which has changed from previous releases.

Determining which server is used to obtain the initial context

WebSphere Application Server name servers are CORBA CosNaming name servers, and WebSphere Application Server provides a CosNaming JNDI plug-in implementation for JNDI clients to perform naming operations on WebSphere Application Server name spaces. The WebSphere Application Server CosNaming plug-in implementation is selected through a JNDI property that is passed to the InitialContext constructor. This property is `java.naming.factory.initial`, and it specifies the initial context factory implementation to use to obtain an initial context. The factory returns a javax.naming.Context instance, which is part of its implementation.

The WebSphere Application Server initial context factory, `com.ibm.websphere.naming.WsnInitialContextFactory`, is typically used by WebSphere Application Server applications to perform JNDI operations. The WebSphere Application Server run-time environment is set up to use this WebSphere Application Server initial context factory if one is not specified explicitly by the JNDI client. When the initial context factory is invoked, an *initial context* is obtained. The following paragraphs explain how the WebSphere Application Server initial context factory obtains the initial context in client and server environments.

- **Understanding the registration of initial references in server processes**

  Every WebSphere Application Server has an ORB used to receive and dispatch invocations on objects running in that server. Services running in the server process can register initial references with the ORB. Each initial reference is registered under a key, which is a string value. An initial reference can be any CORBA object. WebSphere Application Server name servers register several initial contexts as initial references under predefined keys. Each name server initial reference is an instance of the interface org.omg.CosNaming.NamingContext.
- **Obtaining initial references in pure client processes**

  Pure JNDI clients, that is, JNDI clients which are not running in a WebSphere Application Server process, also have an ORB instance. This client ORB instance

can be passed to the InitialContext constructor, but typically the initial context factory creates and initializes the client ORB instance transparently. A client ORB can be initialized with initial references, but the initial references most likely resolve to objects running in some server. The initial context factory does not define any default initial references when it initializes an ORB. If the resolve_initial_references method is invoked on the client ORB when no initial references have been configured, the method invocation fails. This condition is typical for pure client processes. To obtain an initial NamingContext reference, the initial context factory must invoke string_to_object with an IIOP type CORBA object URL, such as corbaloc:iiop:myhost:2809. The URL specifies the address of the server from which to obtain the initial context. The host and port information is extracted from the provider URL passed to the InitialContext constructor. If no provider URL is defined, the WebSphere Application Server initial context factory uses the default provider URL of `corbaloc:iiop:localhost:2809`. The string_to_object ORB method resolves the URL and communicates with the target server ORB to obtain the initial reference.

- **Obtaining initial references in server processes**

  If the JNDI client is running in a WebSphere Application Server process, the initial context factory obtains a reference to the server ORB instance if the JNDI client does not provide an ORB instance. Typically, JNDI clients running in server processes use the server ORB instance; that is, they do not pass an ORB instance to the InitialContext constructor. The name server which is running in the server process sets a provider URL as a java.lang.System property to serve as the default provider URL for all JNDI clients in the process. This default provider URL is `corbaloc:rir:/NameServiceServerRoot`. This URL resolves to the server root context for that server. (The URL is equivalent to invoking resolve_initial_references on the ORB with a key of NameServiceServerRoot. The name server registers the server root context as an initial reference under that key.)

- **Understanding the legacy ORB protocol**

  Previous versions of WebSphere Application Server used a different ORB implementation, which used a legacy protocol in contrast with the Interoperable Name Service (INS) protocol now used. This change has affected the implementation of the WebSphere Application Server initial context factory. **Certain types of pure clients can experience different behavior when getting initial JNDI contexts as compared to previous releases of WebSphere Application Server**. This behavior is discussed in more detail below.

  The following ORB properties are used with the legacy ORB protocol for ORB initialization and are now deprecated:
  - com.ibm.CORBA.BootstrapHost
  - com.ibm.CORBA.BootstrapPort

  The new INS ORB is different in a major respect, in that it exhibits no default behavior if no initial references are defined. In the legacy ORB, the bootstrap host and port values defaulted to localhost and 900. All initial references were obtained from the server running on the bootstrap host and port. So, if the ORB user provided no bootstrap host and port, all initial references are resolved from the server running on the local host at port 900. The INS ORB has no concept of bootstrap host or bootstrap port. All initial references are defined independently. That is, different initial references could resolve to different servers. If ORB.resolve_initial_references is invoked with a key such that the ORB is not initialized with an initial reference having that key, the call fails.

In previous releases of WebSphere Application Server, the initial context factory invoked resolve_initial_references on the ORB in the absence of any provider URL. This action succeeded if a name server at the default bootstrap host and port was running. Today, with the INS ORB, this would fail. (Actually, the ORB would fall back to the legacy protocol during the deprecation period, but when the legacy protocol is no longer supported, the operation would fail.) The initial context factory now uses a default provider URL of `corbaloc:iiop:localhost:2809`, and invokes string_to_object with the provider URL. This operation preserves the behavior that pure clients in previous releases experienced when they set no ORB bootstrap properties or provider URL. **However, this different initial context factory implementation changes the behavior experienced by certain legacy pure clients, which do not specify a provider URL**:

- Clients which set the ORB bootstrap properties listed above when getting an initial context.
- Clients which supply their own ORB instance to the InitialContext constructor.

There are two ways to circumvent this change of behavior:

- Always specify an IIOP type provider URL. This approach does not depend on the bootstrap host and port properties and continues to work when support for the bootstrap host and port properties is removed. For example, you can express bootstrap host and port property values of myHost and 2809, respectively, as corbaloc:iiop:myHost:2809.
- Use an rir type provider URL:
  - Specify `corbaloc:rir:/NameServiceServerRoot` if the ORB is initialized to use a WebSphere Application Server 5 server as the bootstrap server.
  - Specify `corbaname:rir:/NameService#domain/legacyRoot` if the ORB is initialized to use a WebSphere Application Server 4.0.x server as the bootstrap server.
  - Specify `corbaloc:rir:/NameService` if the ORB is initialized to use a server other than a WebSphere Application Server 5 or 4.0.x server as the bootstrap server.

  URLs of this type are equivalent to invoking resolve_initial_references on the ORB with the specified key. If the bootstrap host and port properties are being used to initialize the ORB, this approach will not work when the bootstrap and host properties are no longer supported.

- **The InitialContext constructor search order for JNDI properties**

  If the code snippet shown at the beginning of this section is executed by an application, the bootstrap server depends on the value of the property, java.naming.provider.url. If the property is not set (in server processes the default value is set as a system property), the default host of localhost and default port of 2809 are used as the address of the server from which to obtain the initial context. The JNDI specification describes where the InitialContext constructor looks for java.naming.provider.url property settings, but briefly, the property is picked up from the following places in the order shown:

  1. The InitialContext constructor. This does not apply to the above example since the example uses the empty InitalContext constructor.
  2. System environment. You can add JNDI properties to the system environment as an option on the Java command invocation and by program code. The recommended way to set the provider URL in the system environment is as an option supplied to the Java command invocation. Setting the provider URL in this manner is not temporal, so that getting a default initial context will always yield the same result. It is generally

recommended that program code not set the provider URL property in the system environment because as a side-effect, this could adversely affect other, possibly unrelated, code running elsewhere in the same process.

3. `jndi.properties` file. There may be many `jndi.properties` files that are within the scope of the class loader in effect. All `jndi.properties` files are used for setting JNDI properties, but the provider URL setting is determined by the first jndi.properties file returned by the class loader.

Example: Getting an initial context by setting the provider URL property

In general, JNDI clients should assume the correct environment is already configured so there is no need to explicitly set property values and pass them to the `InitialContext` constructor. However, a JNDI client may need to access a name space other than the one identified in its environment. In this case, it is necessary to explicitly set the java.naming.provider.url (provider URL) property used by the `InitialContext` constructor. A provider URL contains bootstrap server information that the initial context factory can use to obtain an initial context. Any property values passed in directly to the InitialContext constructor take precedence over settings of those same properties found elsewhere in the environment.

You can use two different provider URL forms with WebSphere Application Server's initial context factory:
- A CORBA object URL (new for J2EE 1.3)
- An IIOP URL

CORBA object URLs are more flexible than IIOP URLs and are the recommended URL format to use. CORBA object URLs are part of the OMG CosNaming Interoperable Naming Specification. A corbaname URL, for example, can include initial context and lookup name information and can be used as a lookup name without the need to explicitly obtain another initial context.The IIOP URLs are the legacy JNDI format, but are still supported by the WebSphere Application Server initial context factory.

The following examples illustrate the use of these URLs.

Using a CORBA object URL

This example shows a CORBA object URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL with multiple name server addresses

CORBA object URLs can contain more than one bootstrap address. You can use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap addresses for all servers in the cluster in the URL. The operation succeeds if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address

list will be processed. For example, the second bootstrap address may be used to obtain the initial context even though the server at the first bootstrap address in the list is available.

Multiple-address provider URLs should only contain the bootstrap addresses of members of the same cluster. Otherwise, incorrect behavior may occur.

An example of a corbaloc URL with multiple addresses follows.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.websphere.naming.WsnInitialContextFactory");
// All of the servers in the provider URL below are members of
// the same cluster.
env.put(Context.PROVIDER_URL,
     "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810");
Context initialContext = new InitialContext(env);
...
```

Using a CORBA object URL from an non-WebSphere Application Server JNDI implementation

Initial context factories for CosNaming JNDI plug-in implementations other than the WebSphere Application Server initial context factory most likely obtain an initial context using the object key, `NameService`. When you use such a context factory to obtain an initial context from a WebSphere Application Server name server, the initial context is the cell root context. Since system artifacts such as EJB homes associated with a server are bound under the server's server root context, names used in JNDI operations must be qualified. If you want to use relative names, ensure your initial context is the server root context under which the target object is bound. In order to make the server root context the initial context, specify a corbaloc provider URL with an object key of `NameServiceServerRoot`.

This example shows a CORBA object type URL from a non-WebSphere Application Server JNDI implementation. This example assumes full CORBA object URL support by the non-WebSphere Application Server JNDI implementation. The object key of `NameServiceServerRoot` is specified so that the initial context will be the specified server's server root context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.somecompany.naming.TheirInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaname:iiop:myhost.mycompany.com:9810/NameServiceServerRoot");
Context initialContext = new InitialContext(env);
...
```

If qualified names are used, you can use the default key of `NameService`.

Using an IIOP URL

The IIOP type of URL is a legacy format which is not as flexible as CORBA object URLs. However, URLs of this type are still supported. The following example shows an IIOP type URL as the provider URL.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "iiop://myhost.mycompany.com:2809");
Context initialContext = new InitialContext(env);
...
```

Example: Setting the provider URL property to select a different root context as the initial context

Each server contains its own server root context, and, when bootstrapping to a server, the server root is the default initial JNDI context. Most of the time, this default is the desired initial context, since system artifacts such as EJB homes are bound there. However, other root contexts exist, which can contain bindings of interest. It is possible to specify a provider URL to select other root contexts.

Selecting the initial root context with a CORBA object URL

There are several object keys registered with the bootstrap server that you can use to select the root context for the initial context. To select a particular root context with a CORBA object URL object key, set the object key to the corresponding value. The default object key is NameService. Using JNDI yields the server root context. A table that lists the different root contexts and their corresponding object key follows:

Root Context	CORBA Object URL Object Key
Server Root	NameServiceServerRoot
Cell Persistent Root	NameServiceCellPersistentRoot
Cell Root	NameServiceCellRoot
Node Root	NameServiceNodeRoot

The following example shows the use of a corbaloc URL with the object key set to select the cell persistent root context as the initial context.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL,
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceCellPersistentRoot");
Context initialContext = new InitialContext(env);
...
```

Selecting the initial root context with the name space root property

You can also select the initial root context by passing a name space root property setting to the InitialContext constructor. Generally, the object key setting described above is sufficient. Sometimes a property setting is preferable. For example, you can set the root context property on the Java invocation to make which server root is being used as the initial context transparent to the application. The default server root property setting is `defaultroot`, which yields the server root context.

Root Context	Name Space Root Property Value
Server Root	bootstrapserverroot
Cell Persistent Root	cellpersistentroot
Cell Root	cellroot
Node Root	bootstrapnoderoot

The initial context factory ignores the name space root property if the provider URL contains an object key other than `NameService`.

The following example shows use of the name space root property to select the cell persistent root context as the initial context. Note that available constants are used instead of hard-coding the property name and value.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import com.ibm.websphere.naming.PROPS;
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, "corbaloc:iiop:myhost.mycompany.com:2809");
env.put(PROPS.NAME_SPACE_ROOT, PROPS.NAME_SPACE_ROOT_CELL_PERSISTENT);
Context initialContext = new InitialContext(env);
...
```

Example: Looking up an EJB home with JNDI

Most applications which use JNDI run in a container. Some do not. The name used to look up an object depends on whether or not the application is running in a container. The examples below show lookups from each type of application. Sometimes it is more convenient for an application to use a corbaname URL as the lookup name. Container-based JNDI clients and thin Java clients can use a corbaname URL. An example of a lookup with a corbaname URL is also included in this section.

JNDI lookup from an application running in a container

Applications that run in a container can use `java:` lookup names. Lookup names of this form provide a level of indirection such that the lookup name used to look up an object is not dependent on the object's name as it is bound in the name server's name space. The deployment descriptors for the application provide the mapping from the `java:` name and the name server lookup name. The container sets up the `java:` name space based on the deployment descriptor information so that the `java:` name is correctly mapped to the corresponding object.

The following example shows a lookup of an EJB home. The actual home lookup name is determined by the application's deployment descriptors.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome =
      initialContext.lookup(
         "java:comp/env/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
```

```
}
   catch (NamingException e) { // Error getting the home interface
   ...
}
```

JNDI lookup from an application that does not run in a container

Applications that do not run in a container cannot use `java:` lookup names because it is the container which sets the java: name space up for the application. Instead, an application of this type must look the object up directly from the name server. Each application server contains a name server. System artifacts such as EJB homes are bound relative to the server root context in that name server. The various name servers are federated by means of a system name space structure. The recommended way to look up objects on different servers is to qualify the name so that the name resolves from any initial context in the cell. If a relative name is used, the initial context must be the same server root context as the one under which the object is bound. The form of the qualified name depends on whether the qualified name is a topology-based name or a fixed name. A topology based name depends on whether the object resides in a single server or a server cluster. Examples of each form of qualified name follow.

- **Topology-based qualified names**

  Topology-based qualified names traverse through the system name space to the server root context context under which the target object is bound. A topology-based qualified name resolves from any initial context in the cell. The topology-based qualified name depends on whether the object resides on a single server or server cluster. Examples of each lookup follow.

  Single server

  The following example shows a lookup of an EJB home that is running in the single server, MyServer, configured in the node, Node1.

  ```
  // Get the initial context as shown in a previous example
  // Using the form of lookup name below, it doesn't matter which
  // server in the cell is used to obtain the initial context.
  ...
  // Look up the home interface using the JNDI name
  try {
     java.lang.Object ejbHome = initialContext.lookup(
      "cell/nodes/Node1/servers/MyServer/com/mycompany/accounting
        /AccountEJB");
     accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
        (org.omg.CORBA.Object) ejbHome, AccountHome.class);
  }
  catch (NamingException e) { // Error getting the home interface
     ...
  }
  ```

  Server cluster

  The example below shows a lookup of an EJB home which is running in the cluster, MyCluster. The name can be resolved if any of the cluster members is running.

  ```
  // Get the initial context as shown in a previous example
  // Using the form of lookup name below, it doesn't matter which
  // server in the cell is used to obtain the initial context.
  ...
     // Look up the home interface using the JNDI name
  try {
     java.lang.Object ejbHome = initialContext.lookup(
        "cell/clusters/MyCluster/com/mycompany/accounting/AccountEJB");
     accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
        (org.omg.CORBA.Object) ejbHome, AccountHome.class);
  ```

```
        }
        catch (NamingException e) { // Error getting the home interface
           ...
        }
```

- **Fixed qualified names**

  If the target object has a cell-scoped fixed name defined for it, you can use its qualified form instead of the topology-based qualified name. Even though the topology-based name works, the fixed name does not change with the specific cell topology or with the movement of the target object to a different server. An example lookup with a qualified fixed name is shown below.

```
// Get the initial context as shown in a previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the JNDI name
try {
   java.lang.Object ejbHome = initialContext.lookup(
     "cell/persistent/com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
   }
catch (NamingException e) { // Error getting the home interface
...
}
```

JNDI lookup with a corbaname URL

A corbaname can be useful at times as a lookup name. If, for example, the target object is not a member of the federated name space and cannot be located with a qualifiied name, a corbaname can be a convenient way to look up the object. A lookup with a corbaname URL follows.

```
// Get the initial context as shown in a previous example
...
// Look up the home interface using a corbaname URL
try {
   java.lang.Object ejbHome = initialContext.lookup(
      "corbaname:iiop:someHost:2809#com/mycompany/accounting/AccountEJB");
   accountHome = (AccountHome)javax.rmi.PortableRemoteObject.narrow(
      (org.omg.CORBA.Object) ejbHome, AccountHome.class);
}
catch (NamingException e) { // Error getting the home interface
   ...
}
```

Example: Looking up a JavaMail session with JNDI

The example below shows a lookup of a JavaMail resource. The actual lookup name is determined by the application's deployment descriptors.

```
// Get the initial context as shown above
...
Session session =
      (Session) initialContext.lookup("java:comp/env/mail/MailSession");
```

JNDI interoperability considerations

This section explains considerations to take into account when interoperating with previous releases of WebSphere Application Server and with non-WebSphere Application Server JNDI clients. Also, the way resources from MQSeries must be bound to the name space has changed and is described below.

Interoperability with previous WebSphere Application Server Releases

- **EJB clients running on WebSphere Application Server V3.5 or V4.0 accessing EJB applications running on WebSphere Application Server V5**

  Applications migrated from previous versions of WebSphere Application Server may still have clients still running in a previous release. The default initial JNDI context for EJB clients running on previous versions of WebSphere Application Server is the cell persistent root (legacy root). The home for an enterprise bean deployed in version 5 is bound to its server's server root context. In order for the EJB lookup name for down-level clients to remain unchanged, configure a binding for the EJB home under the cell persistent root.

  Note: EJB clients running in version 3.5 must be running in version 3.5.5 or above, or in version 3.5.3 or 3.5.4 with e-fix PQ51387 installed.
- **EJB clients running on WebSphere Application Server V5 accessing EJB applications running on WebSphere Application Server V3.5 or V4.0 servers**

  The default initial context for a WebSphere Application Server v3.5 or v4.0 server is the correct initial context. Simply look up the JNDI name under which the EJB home is bound.

  Note: To enable WebSphere Application Server V5 clients to access version 3.5.x and 4.0.x servers, the down-level installations must have e-fix PQ60074 installed.

EJB clients running in an environment other than WebSphere Application Server accessing EJB applications running on WebSphere Application Server V5 servers

When an EJB application running in WebSphere Application Server V5 is accessed by a non-WebSphere Application Server EJB client, the JNDI initial context factory is presumed to be a non-WebSphere Application Server implementation. In this case, the default initial context will be the cell root. If the JNDI service provider being used supports CORBA object URLs, the corbaname format can be used to look up the EJB home. The construction of the stringified name depends on whether the object is installed on a single server or cluster, as shown below.

- **Single server**

  ```
  initialContext.lookup(
    "corbaname:iiop:myHost:2809#cell/nodes/node1/servers/server1/myEJB");
  ```

  According to the URL above, the bootstrap host and port are myHost and 2809, and the enterprise bean is installed in a server **server1** in node **node1** and bound in that server under the name **myEJB**.
- **Server cluster**

  ```
  initialContext.lookup(
      "corbaname:iiop:myHost:2809#cell/clusters/myCluster/myEJB");
  ```

  According to the URL above, the bootstrap host and port are **myHost** and **2809**, and the enterprise bean is installed in a server cluster named **myCluster** and bound in that cluster under the name **myEJB**.

  The above lookup will work with any name server bootstrap host and port configured in the same cell.

The above lookup will also work if the bootstrap host and port belongs to a member of the cluster itself. To avoid a single point of failure, the bootstrap server host and port for each cluster member could be listed in the URL as follows:

```
initialContext.lookup(
    "corbaname:iiop:host1:9810,host2:9810#cell/clusters/myCluster/myEJB");
```

The name prefix **cell/clusters/myCluster/** is not necessary if boostrapping to the cluster itself, but it will work. The prefix is needed, however, when looking up enterprise beans in other clusters. Name bindings under the **clusters** context are implemented on the name server to resolve to the server root of a running cluster member during a lookup; thus avoiding a single point of failure.

- **Without CORBA object URL support**

  If the JNDI initial context factory being used does not support CORBA object URLs, the initial context can be obtained from the server, and the lookup can be performed on the initial context as follows:

```
Hashtable env = new Hashtable();
env.put(CONTEXT.PROVIDER_URL, "iiop://myHost:2809");
Context ic = new InitialContext(env);
Object o = ic.lookup("cell/clusters/myCluster/myEJB");
```

Binding resources from MQSeries 5.2

In previous releases of WebSphere Application Server, the MQSeries jmsadmin tool could be used bind resources to the name space. When used with a WebSphere Application Server V5 name space, the resource will be bound within a transient partition in the name space and will not persist past the life of the server process. Instead of binding the MQSeries resources with the jmsadmin tool, bind them from the WebSphere Application Server administrative console, under Resources in the left panel on the console

JNDI caching

To increase the performance of JNDI operations, the WebSphere Application Server JNDI implementation employs caching to reduce the number of remote calls to the name server for lookup operations. For most cases, use the default cache setting.

When an InitialContext object is instantiated, an association is established between the InitialContext instance and a cache. The initial context and any contexts returned directly or indirectly from a lookup on the initial context are all associated with that same cache instance. By default, the association is based on the provider URL, in particular, the host name and port. The caller can specify the cache name to override this default behavior. A cache instance of a given name is shared by all instances of InitialContext configured to use a cache of that name which were created with the same context class loader in effect. Two EJB applications running in the same server will use their own cache instances, if they are using different context class loaders, even if the cache names are the same.

After an association between an InitialContext instance and cache is established, the association does not change. A `javax.naming.Context` object returned from a lookup operation inherits the cache association of the Context object on which the lookup was performed. Changing cache property values with the `Context.addToEnvironment()` or `Context.removeFromEnvironment()` method does not affect cache behavior. You can change properties affecting a given cache instance with each InitialContext instantiation.

A cache is restricted to a process and does not persist past the life of that process. A cached object is returned from lookup operations until either the max cache life for the cache is reached, or the max entry life for the object's cache entry is reached.

After this time, a lookup on the object causes the cache entry for the object to be refreshed. If a bind or rebind operation is executed on an object, the change is not reflected in any caches other than the one associated with the context from which the bind or rebind was issued. This scenario is most likely to happen when multiple processes are involved, since different processes do not share the same cache, and context objects in all threads in a process typically share the same cache instance for a given name service provider.

Usually, cached objects are relatively static entities, and objects becoming stale are not a problem. However, you can set timeout values on cache entries or on a cache so that cache contents are periodically refreshed.

JNDI cache settings

Various cache property settings follow. Ensure that all property values are string values.

com.ibm.websphere.naming.jndicache.cachename

The name of the cache to associate with an initial context instance can be specified with this property.

It is possible to create multiple InitialContext instances, each operating on the name space of a different name server. By default, objects from each bootstrap address are cached separately, since they each involve independent name spaces and name collisions could occur if they used the same cache. The provider URL specified when the initial context is created by default serves as the basis for the cache name. With this property, a JNDI client can specify a cache name. Valid options for cache names follow:

Valid options	Resulting cache behavior
providerURL (default)	Use the value for java.naming.provider.url property as the basis for the cache name. Cache names are based on the bootstrap host and port specified in the URL. The boostrap host is normalized to a fully qualfied name, if possible. For example, ″corbaname:iiop:server1:2809#some/starting/context″ and ″corbaloc:iiop://server1″ are normalized to the same cache name. If no provider URL is specified, a default cache name is used.
Any string	Use the specified string as the cache name. You can use any arbitrary string with a value other than ″providerURL″ as a cache name.

com.ibm.websphere.naming.jndicache.cacheobject

Turn caching on or off and clear an existing cache with this property.

By default, when an InitialContext is instantiated, it is associated with an existing cache or, if one does not exist, a new one is created. An existing cache is used with its existing contents. In some circumstances, this behavior is not desirable. For example, when objects that are looked up change frequently, they can become stale in the cache. Other options are available. Thefollowing table lists these other options along with the corresponding property value.

Valid values	Resulting cache behavior
populated (default)	Use a cache with the specified name. If the cache already exists, leave existing cache entries in the cache; otherwise, create a new cache.
cleared	Use a cache with the specified name. If the cache already exists, clear all cache entries from the cache; otherwise, create a new cache.
none	Do not cache. If this option is specified, the cache name is irrelevant. Therefore, this option will not disable a cache that is already associated with other InitialContext instances. The InitialContext that is instantiated is not associated with any cache.

com.ibm.websphere.naming.jndicache.maxcachelife

Impose a limit to the age of a cache with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to "cleared". This property enables a JNDI client to set the maximum life of a cache. This property differs from the maxentrylife property (below) in that the entire cache is cleared when the cache lifetime is reached. The table below lists the various maxcachelife values and their affect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Make the cache lifetime unlimited.
Positive integer	Set the maximum lifetime of the entire cache, in minutes, to the specified value. When the maximum lifetime for the cache is reached, the next attempt to read any entry from the cache causes the cache to be cleared

com.ibm.websphere.naming.jndicache.maxentrylife

Impose a limit to the age of individual cache entries with this property.

By default, cached objects remain in the cache for the life of the process or until cleared with the com.ibm.websphere.naming.jndicache.cacheobject property set to `cleared`. This property enables a JNDI client to set the maximum lifetime of individual cache entries. This property differs from the maxcachelife property in that individual entries are refreshed individually as their maximum lifetime reached. This might avoid any noticeable change in performance that might occur if the whole cache is cleared at once. The table below lists the various maxentrylife values and their effect on cache behavior:

Valid options	Resulting cache behavior
0 (default)	Lifetime of cache entries is unlimited.
Positive integer	Set the maximum lifetime of individual cache entries, in minutes, to the specified value. When the maximum lifetime for an entry is reached, the next attempt to read the entry from the cache causes the individual cache entry to refresh.

Example: Controlling JNDI cache behavior from a program

Following are examples that illustrate how you can use JNDI cache properties to achieve the desired cache behavior. Cache properties take effect when an InitialContext object is constructed.

```
import java.util.Hashtable;
import javax.naming.InitialContext;
import javax.naming.Context;
import com.ibm.websphere.naming.PROPS;

/*****
 Caching discussed in this section pertains to the WebSphere Application Server
 initial context factory. Assume the property, java.naming.factory.initial,
 is set to "com.ibm.websphere.naming.WsnInitialContextFactory" as a
 java.lang.System property.
*****/

Hashtable env;
Context ctx;

// To clear a cache:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_CLEARED);
ctx = new InitialContext(env);

// To set a cache's maximum cache lifetime to 60 minutes:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_MAX_LIFE, "60");
ctx = new InitialContext(env);

// To turn caching off:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
ctx = new InitialContext(env);

// To use caching and no caching:

env = new Hashtable();
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_POPULATED);
ctx = new InitialContext(env);
env.put(PROPS.JNDI_CACHE_OBJECT, PROPS.JNDI_CACHE_OBJECT_NONE);
Context noCacheCtx = new InitialContext(env);

Object o;

// Use caching to look up home, since the home should rarely change.
o = ctx.lookup("com/mycom/MyEJBHome");
// Narrow, etc. ...

// Do not use cache if data is volatile.
o = noCacheCtx.lookup("com/mycom/VolatileObject");
// ...
```

JNDI name syntax

JNDI name syntax is the default syntax and is suitable for typical JNDI clients.

This syntax includes the following special characters: forward slash (/) and backslash (\). Components in a name are delimited by a forward slash. The backslash is used as the escape character. A forward slash is interpreted literally if it is escaped, that is, preceded by a backslash. Similarly, a backslash is interpreted literally if it is escaped.

INS name syntax

INS syntax is designed for JNDI clients that need to interoperate with CORBA applications.

The INS syntax allows a JNDI client to make the proper mapping to and from a CORBA name. INS syntax is very similar to the JNDI syntax with the additional special character, dot (.). Dots are used to delimit the `id` and `kind` fields in a name component. A dot is interpreted literally when it is escaped. Only one unescaped dot is allowed in a name component. A name component with a non-empty `id` field and empty `kind` field is represented with only the `id` field value and must not end with an unescaped dot. An empty name component (empty `id` and empty `kind` field) is represented with a single unescaped dot. An empty string is not a valid name component representation.

JNDI to CORBA name mapping considerations

WebSphere Application Server name servers are an implementation of the CORBA CosNaming interface. WebSphere Application Server provides a JNDI implementation which you can use to access CosNaming name servers through the JNDI interface. Issues can exist when mapping JNDI name strings to and from CORBA names.

Each component in a CORBA name consists of an `id` and `kind` field, but a JNDI name component consists of no such fields. Each component in a JNDI name is atomic. Typical JNDI clients do not need to make a distinction between the `id` and `kind` fields of a name component, or know how JNDI name strings map to CORBA names. JNDI clients of this sort can use the JNDI syntax described below. When a name is parsed according to JNDI syntax, each name component is mapped to the `id` field of the corresponding CORBA name component. The `kind` field always has an empty value. This basic syntax is the least obtrusive to the JNDI client in that it has the fewest special characters. However, you cannot represent with this syntax a CORBA name with a non-empty `kind` field. This restriction can prevent EJB applications from interoperating with CORBA applications.

Some clients, however must interoperate with CORBA applications which use CORBA names with non-empty `kind` fields. These JNDI clients must make a distinction between `id` and `kind` so that JNDI names are correctly mapped to CORBA names, particularly when the CORBA names contain components with non-null `kind` fields. Such JNDI clients can use the INS name syntax. With its additional special character, you can use INS to represent any CORBA name. Use of this syntax is not recommended unless it is necessary, because this syntax is more restrictive from the JNDI client's perspective in that the JNDI client must be aware that name components with multiple unescaped dots are syntactically invalid. INS name syntax is part of the OMG CosNaming Interoperable Naming Specification.

Example: Setting the syntax used to parse name strings

JNDI clients which must interoperate with CORBA applications may need to use INS name syntax to represent names in string format. The name syntax property may be passed to the InitialContext constructor through its parameter, in the System properties, or in a jndi.properties file. The initial context and any contexts looked up from that initial context will parse name strings based on the specified syntax.

The following example shows how to set the name syntax to make the initial context parse name strings according to INS syntax.

```
...
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import com.ibm.websphere.naming.PROPS; // WebSphere naming constants
...
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");
env.put(Context.PROVIDER_URL, ...);
env.put(PROPS.NAME_SYNTAX, PROPS.NAME_SYNTAX_INS);
Context initialContext = new InitialContext(env);
// The following name maps to a CORBA name component as follows:
//    id = "a.name", kind = "in.INS.format"
// The unescaped dot is used as the delimiter.
// Escaped dots are interpreted literally.
java.lang.Object o = initialContext.lookup("a\.name.in\.INS\.format");
...
```

Developing applications that use CosNaming (CORBA Naming interface)

CORBA clients can perform naming operations on WebSphere name servers through the CosNaming interface. The following examples show how to obtain an ORB instance and an initial context as well as how to look up an EJB home.

Note: To enable WebSphere Application Server V5 clients to access Versions 3.5.x and 4.0.x servers, the earlier installations must have e-fix PQ60074 installed.

1. Get an initial context
2. Perform desired CosNaming operations

Example: Getting an initial context with CosNaming

In the WebSphere Application Server, an initial context is obtained from a bootstrap server. The address for the bootstrap server consists of a host and port. To get an initial context, you must know the host and port for the server that is used as the bootstrap server.

Obtaining an initial context consists of two basic steps:
1. Obtain an ORB reference
2. Invoke a method on the ORB to obtain the initial reference

These steps are now explained in more detail.

Obtaining an ORB reference

Pure CosNaming clients, that is clients that are not running in a server process, must create and initialize an ORB instance with which to obtain the initial context. CosNaming clients which run in server processes can obtain a reference to the server ORB with a JNDI lookup. The following examples illustrate how to create and initialize a client ORB and how to obtain a server ORB reference.

Creating a client ORB instance

To create an ORB instance, invoke the static method, org.omg.CORBA.ORB.init. The init method requires a property set to the name of the ORB class you want to instantiate. An ORB implementation with the class name com.ibm.CORBA.iiop.ORB is included with the WebSphere Application Server. The WebSphere Application Server ORB recognizes additional properties with which you can specify initial references.

The basic steps for creating an ORB are as follows:

1. Create a Properties object.
2. Set the ORB class property to WebSphere Application Server's ORB class.
3. If the bootstrap server is INS-compliant, set the initial reference properties. If the bootstrap server is not INS-compliant (meaning, WebSphere Application Server v4.0.x or earlier), set bootstrap host and port for bootstrap server.
4. Invoke ORB.init, passing in the Properties object.

Usage scenario

```
...
import java.util.Properties;
import org.omg.CORBA.ORB;
...
Properties props = new Properties();
props.put("org.omg.CORBA.ORBClass","com.ibm.ws390.orb.ORB");
props.put("com.ibm.CORBA.ORBInitRef.NameService",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameService");
props.put("com.ibm.CORBA.ORBInitRef.NameServiceServerRoot",
      "corbaloc:iiop:myhost.mycompany.com:2809/NameServiceServerRoot");
// props.put("com.ibm.CORBA.BootstrapHost", "myhost.mycompany.com");
// Use this if bootstrap server is WebSphere 4.0.x or before
// props.put("com.ibm.CORBA.BootstrapPort", "2809");
// Use this if bootstrap server is WebSphere 4.0.x or before
ORB _orb = ORB.init((String[])null, props);
...
```

Notice the initial reference definitions for NameService and NameServiceServerRoot. The initial context returned for NameService depends on the type of bootstrap server. The key NameServiceServerRoot is a key introduced in WebSphere Application Server V5. For more information on initial contexts, see the section Initial Contexts.

Note: The properties com.ibm.CORBA.BootstrapHost and com.ibm.CORBA.BootstrapPort are deprecated. They are needed, however, to connect to WebSphere Application Servers of Version 4.0.x or earlier. The default bootstrap host is the local host and the default port is 2809.

Obtaining a reference to the server ORB

CosNaming clients which run in a server process can obtain a reference to the server ORB with a JNDI lookup on a java: name, shown as follows:

Usage scenario

```
...
import javax.naming.Context;
import javax.naming.InitialContext;
import org.omg.CORBA.ORB;
...
Context initialContext = new InitialContext();
ORB orb = (ORB) initialContext.lookup("java:comp/ORB");
...
```

Using an ORB reference to get an initial naming reference

There are two basic ways to get an initial CosNaming context. Both ways involve an ORB method invocation. The first way is to invoke the resolve_initial_references method on the ORB with an initial reference key. For this call to work, the ORB must be initialized with an initial reference for that key. The other way is to invoke the string_to_object method on the ORB, passing in a CORBA object URL with the host and port of the bootstrap server. The following examples illustrate both approaches.

Invoking resolve_initial_references

Once an ORB reference is obtained, invoke the resolve_initial_references method on the ORB to obtain a reference to the initial context. The following code example invokes resolve_initial_reference on an ORB reference.

Usage scenario

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
org.omg.CORBA.Object obj = _orb.resolve_initial_references("NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is passed to the resolve_initial_references method. Other initial context keys are registered in WebSphere Application Servers. For example, NameServiceServerRoot can be used to obtain a reference to the server root context in the bootstrap name server. For more information on the initial contexts registered in server ORBs, please see the section Initial Contexts.

Invoking string_to_object with a CORBA object URL

You can use an INS-compliant ORB to obtain an initial context even if the ORB is not initialized with any initial references or bootstrap properties, or if those property settings are for a different server than the name server from which you want to obtain the initial context. To obtain an initial context by explicitly specifying the bootstrap name server, invoke the string_to_object method on the ORB, passing in a CORBA object URL which contains the bootstrap server host and port.

The code in the example below invokes the string_to_object method on an existing ORB reference, passing in a CORBA object URL which identifies the desired initial context.

Usage scenario

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Obtain ORB reference as shown in examples earlier in this section
...
org.omg.CORBA.Object obj =
    orb.string_to_object("corbaloc:iiop:myhost.mycompany.com:2809/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Note that the key NameService is used in the corbaloc URL. Other initial context keys are registered in WebSphere Application Servers. For example, you can use NameServiceServerRoot to obtain a reference to the server root context in the bootstrap name server.

Using an existing ORB and invoking string_to_object with a CORBA object URL with multiple name server addresses to get an initial context

CORBA object URLs can contain more than one bootstrap server address. Use this feature when attempting to obtain an initial context from a server cluster. You can specify the bootstrap server addresses for all servers in the cluster in the URL. The operation will succeed if at least one of the servers is running, eliminating a single point of failure. There is no guarantee of any particular order in which the address list will be processed. For example, the second bootstrap server address may be used to obtain the initial context even though the first bootstrap server in the list is available. An example of a corbaloc URL with multiple addresses follows.

```
...
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
...
// Assume orb is an existing ORB instance
org.omg.CORBA.Object obj = orb.string_to_object(
   "corbaloc::myhost1:9810,:myhost1:9811,:myhost2:9810/NameService");
NamingContextExt initCtx = NamingContextExtHelper.narrow(obj);
...
```

Example: Looking up an EJB home with CosNaming

You can look up an EJB home or other CORBA object from a WebSphere Application Server name server through the CORBA CosNaming interface. You can invoke `resolve` or `resolve_str` on the initial context, or you can invoke string_to_object on the ORB. You can use a qualified name so that the name resolves regardless of which name server the lookup is executed on, or use an unqualified name that only resolves from the server root context on the name server that actually contains the object binding. (The qualified name traverses the federated system name space to the specified server root context.)

Qualified and unqualified names

Each application server contains a name server. System artifacts such as EJB homes are bound in that name server. The various name servers are federated by means of a system name space structure. The recommended way to look up objects on different servers is to use a qualified name. A qualified name can be a topology-based name, based on the name of the cluster or single server and node that contains the object. You can define fixed qualified names for objects. With qualified names, you can look up objects residing on different servers from the same initial context by traversing the system name space structure. Alternatively, you can use an unqualified name, but an unqualified name will only resolve using the name server associated with the object's application server.

CosNaming.resolve (and resolve_str) vs. ORB.string_to_object

If you have an initial context from any name server in a WebSphere Application Server cell, you can look up any CORBA object with a qualified name. You do not need additional host and port information for the target object's name server.

Alternatively, you can look up an object by invoking string_to_object on the ORB, passing in a corbaname URL. Typically, an IIOP type URL is specified, so the bootstrap address information required for an initial context must be contained in the URL. You can use a qualified or unqualified stringified name, but an unqualifed name resolves only if the initial context is from the name server in which the object is bound.

The following examples show CosNaming resolve operations using qualified topology-based lookup names and an unqualified lookup name.

CosNaming resolve operation using a qualified name

The topology-based qualified name for an object depends on whether the object is bound in a single server or a server cluster. Examples of each follow.

Single Server

The following example shows the lookup of an EJB home that is running in a single server. The enterprise bean that is being looked up is running in the server, `MyServer`, on the node, `Node1`.

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
   "cell/nodes/Node1/servers/MyServer/mycompany/accounting/AccountEJB");
accountHome =
   (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Server Cluster

The following example shows a lookup of an EJB home that is running in a cluster. The enterprise bean being that is looked up is running in the cluster, Cluster1. The name can be resolved if any of the cluster members is running.

Usage scenario

```
// Get the initial context as shown in the previous example
// Using the form of lookup name below, it doesn't matter which
// server in the cell is used to obtain the initial context.
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = initialContext.resolve_str(
   "cell/clusters/Cluster1/mycompany/accounting/AccountEJB");
accountHome =
   (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

ORB string_to_object operation using an unqualified stringified name

If the resolve operation is being performed on the name server that contains the object, the system name space does not need to be traversed, and you can use an unqualified lookup name. Note that this name does not resolve on other name servers. If an unqualified name is provided, the object key must be NameServiceServerRoot so that the correct initial context is selected. If a qualified name is provided, you can use the default key of NameService.

The following example shows a lookup of an EJB home. The enterprise bean that is being looked up is bound on the name server running on the host `myHost` on port `2809`. Note the object key of `NameServiceServerRoot`.

```
// Assume orb is an existing ORB instance
...
// Look up the home interface using the name under which the EJB home is bound
org.omg.CORBA.Object ejbHome = orb.string_to_object(
    "corbaname:iiop:myHost:2809/NameServiceServerRoot#mycompany/accounting");
accountHome =
    (AccountHome)javax.rmi.PortableRemoteObject.narrow(ejbHome, AccountHome.class);
```

Configured name bindings

Administrators can configure bindings into the name space. A configured binding is different from a programmatic binding in that the system creates the binding every time a server is started, even if the target context is in a transient partition.

Administrators can add name bindings to the name space through the configuration. Name servers add these configured bindings to the name space view, by reading the configuration data for the bindings. Configuring bindings is an alternative to creating the bindings from a program. Configured bindings have the advantage of being created each time a server starts, even when the binding is created in a transient partition of the name space. Cell-scoped configured bindings provide interoperability with JNDI clients running on previous versions of WebSphere Application Server. Additionally, you can configure cell-scoped bindings to create a fixed qualified name for server objects.

Scope

You can configure a binding at one of the following three scopes: cell, node, or server. Cell-scoped bindings are created under the cell persistent root context. Node-scoped bindings are created under the node persistent root context for the specified node. Server-scoped bindings are created under the server root context for the selected server. If the target server of a server-scoped binding is a cluster, the binding is created under the server root context of each cluster member.

Note: The term *server* includes clusters and can be used interchangeably with the term *cluster* with respect to configured bindings. When applied to a cluster, a server-scoped binding is created in the server root for all member servers.

The scope you select for new bindings depends on how the binding is to be used. For example, if the binding is not specific to any particular node or server, or if you do not want the binding to be associated with any specific node or server, a cell-scoped binding is a suitable scope. Defining fixed names for enterprise beans to create fixed qualified names is just such an application. If a binding is to be used only by clients of an application running on a particular server, or if you want to configure a binding with the same name on different servers which resolve to different objects, a server-scoped binding would be appropriate. Note that two servers can have configured bindings with the same name but resolve to different objects. At the cell scope, only one binding with a given name can exist.

Intermediate Contexts

Intermediate contexts created with configured bindings are read-only. For example, if an EJB home binding is configured with the name `some/compound/name/ejbHome`, the intermediate contexts `some`, `some/compound`, and `some/compound/name` will be created as read-only contexts. You cannot add, update, or remove any read-only bindings.

The configured binding name cannot conflict with existing bindings. However, configured bindings can use the same intermediate context names. Therefore, a configured binding with the name `some/compound/name2/ejbHome2` does not conflict with the previous example name.

Configured binding types

Types of objects that you can bind follow:

EJB: EJB home installed in some server in the cell

The following data is required to configure an EJB home binding:
- JNDI name of the EJB server or server cluster where the enterprise bean is deployed
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root.

This type of binding is of special significance because you can use it to provide interoperability with WebSphere Application Server v3.5.x and v4.0.x JNDI clients. The default initial context for these earlier clients is the cell persistent root, which is different from the initial context of the server root for WebSphere Application Server V5 JNDI clients. If you migrate an application to the current release, you can configure an EJB binding at the cell scope so that the lookup names for the enterprise bean do not change for clients still running in a earlier WebSphere Application Server version.

A cell-scoped EJB binding is also useful for creating a fixed lookup name for an enterprise bean so that the qualified name is not dependent on the topology.

CORBA: CORBA object available from some CosNaming name server

You can identify any CORBA object bound into some INS compliant CosNaming server with a corbaname URL. The referenced object does not have to be available until the binding is actually referenced by some application.

The following data is required in order to configure a CORBA object binding:
- The corbaname URL of the CORBA object
- An indicator if the bound object is a context or leaf node object (to set the correct CORBA binding type of context or object).
- Target root for the configured binding
- The name of the configured binding, relative to the target root.

Indirect: Any object bound in WebSphere Application Server name space accessible with JNDI

Besides CORBA objects, this includes javax.naming.Referenceable, javax.naming.Reference, and java.io.Serializable objects. The target object itself is not bound to the name space. Only the information required to look up the object is bound. Therefore, the referenced name server does not have to be running until the binding is actually referenced by some application. The following data is required in order to configure an indirect JNDI lookup binding:
- JNDI provider URL of name server where object resides
- JNDI lookup name of object
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root.

A cell-scoped indirect binding is useful when creating a fixed lookup name for a resource so that the qualified name is not dependent on the topology. You can also achieve this topology by widening the scope of the resource definition.

Note: WebSphere Application Server v3.5.x clients cannot access this type of binding .

String: String constant

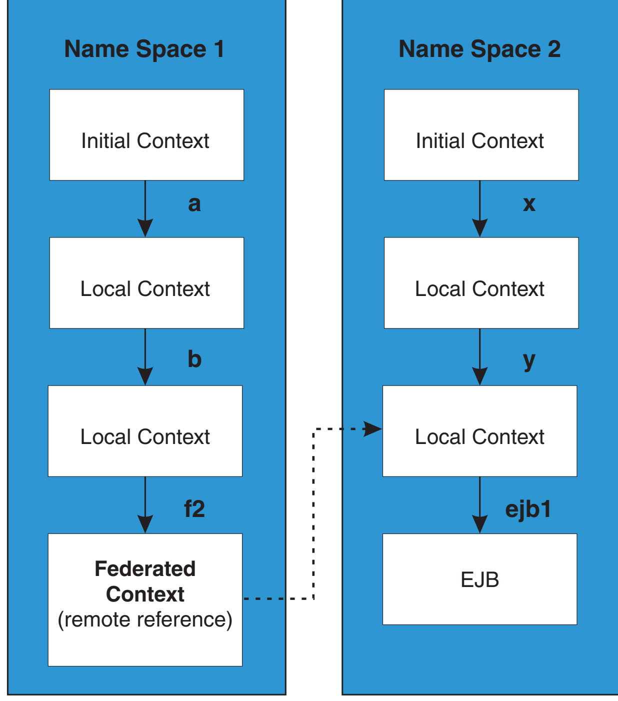
You can configure a binding of a string constant. The following data is required to configure a string constant binding:

- String constant value
- Target root for the configured binding (scope)
- The name of the configured binding, relative to the target root.

Name space federation

Federating name spaces involves binding contexts from one name space into another name space.

For example, assume that a name space, Name Space 1, contains a context under the name a/b. Also assume that a second name space, Name Space 2, contains a context under the name x/y. (See the following illustration.) If context x/y in Name Space 2 is bound into context a/b in Name Space 1 under the name f2, the two name spaces are federated. Binding f2 is a federated binding because the context associated with that binding comes from another name space. From Name Space 1, a lookup of the name a/b/f2 returns the context bound under the name x/y in Name Space 2. Furthermore, if context x/y contains an Enterprise JavaBeans (EJB) home bound under the name ejb1, the EJB home could be looked up from Name Space 1 with the lookup name a/b/f2/ejb1. Notice that the name crosses name spaces. This fact is transparent to the naming client.



In a WebSphere Application Server name space, you can create federated bindings with the following restrictions:

- Federation is limited to CosNaming name servers. A WebSphere Application Server name server is a Common Object Request Broker Architecture (CORBA) CosNaming implementation. You can create federated bindings to other CosNaming contexts. You cannot, for example, bind contexts from an LDAP name server implementation.

- If you use JNDI to federate the name space, you must use WebSphere Application Server's initial context factory to obtain the reference to the federated context. If you use some other initial context factory implementation, you either may not be able to create the binding, or the level of transparency may be reduced.
- A federated binding to a non-WebSphere Application Server naming context has the following functional limitations:
  - JNDI operations are restricted to the use of CORBA objects. For example, you can look up EJB homes, but you cannot look up non-CORBA objects such as data sources.
  - JNDI caching is not supported for non-WebSphere Application Server name spaces. This restriction affects the performance of lookup operations only.
- Do not federate two WebSphere Application Server stand-alone server name spaces. Incorrect behavior may result. If you want to federate WebSphere Application Server name spaces, you should use servers running under the Network Deployment or Enterprise packages of WebSphere Application Server.

Name space bindings

Administrators can add name bindings to the name space through the configuration. Name servers add these configured bindings to the name space view by reading the configuration data for the bindings. Configuring bindings is an alternative to creating the bindings from a program.

Configured bindings are created each time a server starts, even when the binding is created in a transient partition of the name space. One major use of configured bindings to provide interoperability with JNDI clients running on previous versions of the WebSphere Application Server.

There are four different kinds of bindings that you can configure:
- Enterprise JavaBeans (EJB)
- CORBA object
- Indirect Lookup
- String

Configuring and viewing name space bindings

To view or configure an EJB, CORBA, Indirect lookup or string name space binding, complete the following:

1. Open the Administrative console.
2. Click **Environment**.
3. Click **Manage Name Space Bindings**.
4. Select the desired scope by entering in a node name for node-scoped bindings, or a node name and server name for server-scoped bindings, and click **Apply**.
5. To create a new binding, click New and follow the instructions. To edit a previously created binding, click the binding you want to edit and proceed to the next step.
6. Edit the Binding identifier, the Name in name space, and the String value fields as desired.

   Note: All of these fields are required.
7. Click **Finish** to register the changes.

String binding settings

Use this page to configure a new string binding or to view or edit an existing string binding.

To view this administrative console page, click **Environment** > **Naming** > **Name Space Bindings** > *string_namespace_binding*.

Scope

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier

Specifies the name that uniquely identifies this configured binding.

Name in Name Space

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

String Value

Specifies the string to be bound into the name space.

CORBA object binding settings

Use this page to configure a new name binding of a CORBA object binding, or to view or edit an existing CORBA object binding.

To view this administrative console page, click **Environment** > **Naming** > **Name Space Bindings** > *CORBA_namespace_binding*.

Scope

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier

Specifies the name that uniquely identifies this configured binding.

Name in Name Space

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Corbaname URL

Specifies the CORBA name URL string identifying where the object is bound in a CosNaming server.

Federated Context

Specifies whether the target is a CosNaming context (true) or a leaf node object (false).

Value	Result
true	The target object is bound with a context CORBA binding type. If the corbaname URL does not resolve to a NamingContext, an error occurs when the binding is first used (which is when the URL is first resolved).
false	The target object is bound with an object CORBA binding type.

Indirect lookup binding settings

Use this page to configure a new indirect lookup name binding, or to view or edit an existing indirect lookup binding.

To view this administrative console page, click **Environment** > **Naming** > **Name Space Bindings** > *indirect_lookup_namespace_binding*.

Scope

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier

Specifies the name that uniquely identifies this configured binding.

Name in Name Space

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Provider URL

Specifies the provider URL string needed to obtain a JNDI initial context.

JNDI Name

Specifies the name used to look up the target object from the initial context.

EJB binding settings

Use this page to configure a new EJB binding, or to view or edit an existing EJB binding.

To view this administrative console page, click **Environment** > **Naming** > **Name Space Bindings** > *EJB_namespace_binding*.

Scope

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type

Shows the type of binding configured. Possible choices are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Binding Identifier

Specifies the name that uniquely identifies this configured binding.

Name in Name Space

Specifies the name used for this binding in the name space. This name can be a simple or compound name depending on the portion of the name space where this binding is configured.

Enterprise Bean Location

Specifies whether the enterprise bean is running in a server cluster or a single server. If Single Server is specified, type the node name.

Server

Specifies the name of the cluster or non-clustered server in which the enterprise bean is configured.

JNDI Name

Specifies the JNDI name of the deployed enterprise bean (the bean's JNDI name that is in the enterprise bean bindings--not the java:comp name)

Name space binding collection

Use this page to configure a name binding of an EJB, a CORBA CosNaming NamingContext, a CORBA leaf node object, an object that you can look up using JNDI, or a constant string value.

Binding information for configured bindings is stored in the configuration and applied upon startup of the name server for each server within the scope of the binding.

To view the Manage Name Space Bindings Settings page, click **Environment > Naming > Name Space Bindings**.

Click the check boxes to select one or more of the users in your collection. Use the buttons to control the selected users.

Name

Shows the names given to uniquely identify these configured bindings.

Scope

Shows the scope of the configured binding. This value indicates the configuration location for the `namebindings.xml` file. This field is for information purposes only and cannot be updated.

If the configured binding is cell-scoped, the starting context is the cell persistent root context. If the configured binding is node-scoped, the starting context is the node persistent root context. If the configured binding is server-scoped, the starting context is the server's server root context.

Binding Type

Shows the type of binding configured. Valid values are String, EJB, CORBA, and Indirect. This field is for information purposes only and cannot be updated.

Configuring name servers

To configure a name server, complete the following:

1. Open the administrative console.
2. Click **Servers**.
3. Click **Application Servers**.
4. Click the application server you want to configure.
5. Click **Server Components**.
6. Click **Name Server**.
7. Edit the fields as desired.

   Note: All of these fields are mandatory.
8. To make other changes, click **Custom Properties**.
9. Click **OK** to register your changes.

Name server settings

Use this page to configure Naming Service Provider settings for the application server.

To view this administrative console page, click one of the following paths:

- **Servers > Application Servers >***server_name* **> Server Components > Name Server**
- **Servers > JMS Servers >***server_name* **> Server Components > Name Server**

Name

Specifies the display name for the server.

Data type	String

Initial State

Specifies the execution state. The options are: *Started* and *Stopped*.

Data type	String
Default	Started

Troubleshooting name space problems

Many naming problems can be avoided by fully understanding the key underlying concepts of WebSphere Application Server naming.

1. Review the key concepts of WebSphere Application Server naming, especially Name space logical view and Lookup names support in deployment descriptors and thin clients.

2. Review the programming examples that are included in the sections explaining the JNDI and CosNaming interfaces.
3. Read Naming services component troubleshooting tips for additional general information.
4. If you Cannot look up an object hosted by WebSphere Application Server from a servlet, JSP file, or other client, read this article.

dumpNameSpace tool

You can use the dumpNameSpace tool to dump the contents of a name space accessed through a name server. When you invoke the dumpNameSpace tool, the naming service must be active. The dumpNameSpace tool cannot dump name spaces local to the server process, such as those with java: and local: URL schemes. The local: name space contains references to enterprise beans with local interfaces. Use the name space dump utility for java:, local: and server name spaces to dump java: and local: name spaces.

Note that the server root context for the server at the specified host and port is dumped (unless a non-default starting context which precludes it is specified). The server root contexts for other servers are not dumped.

If you run the dumpNameSpace tool, a login prompt is displayed. If you cancel the login prompt, the dumpNameSpace tool continues outbound with an ″UNAUTHENTICATED″ credential. Thus, by default, an ″UNAUTHENTICATED″ credential is used that is equivalent to the ″Everyone″ access authorization policy. You can modify this default setting by changing the value for the `com.ibm.CSI.performClientAuthenticationRequired` property to `true` in the *install_dir*`/properties/sas.client.props` file. When you change this property to `true`, rerun the dumpNameSpace tool, and cancel the login prompt, the authorization fails and the command will not continue outbound.

Command line invocation descriptions of the dumpNameSpace tool follow. This section includes sample output.

You can also access this tool a through its program interface. Refer to the class com.ibm.websphere.naming.DumpNameSpace in the WebSphere Application Server API documentation.
To invoke the tool through the command line, enter the following command from the *WebSphere/AppServer/bin* directory:

Platform	Command
UNIX	dumpNameSpace.sh [[-*keyword value*]...]
Windows NT	dumpNameSpace [[-*keyword value*]...]

Parameters

The keywords and associated values for the dumpNameSpace utility follow:

-host *myhost.austin.ibm.com*
Indicates the bootstrap host or the WebSphere Application Server host whose name space you want to dump. The value defaults to ***localhost***.

-port *nnn*
Indicates the bootstrap port which, if not specified, defaults to **2809**.

-root {cell | server | node | host | legacy | tree | default}
Indicates the root context to use as the initial context for the dump. The applicable root options and default root context depend on the type of name server from which the dump is being obtained. This information is provided in the following tables.

For WebSphere Application Servers V5 or later:

cell	DumpNameSpace default. Dump the tree starting at the cell root context.
server	Dump the tree starting at the server root context.
node	Dump the tree starting at the node root context. (Synonymous with host.)

For WebSphere Application Servers v4.0 or later:

legacy	DumpNameSpace default. Dump the tree starting at the legacy root context.
host	Dump the tree starting at the bootstrap host root context. (Synonymous with node.)
tree	Dump the tree starting at the tree root context.

For all WebSphere Application Servers and other name servers:

default	Dump the tree starting at the initial context which JNDI returns by default for that server type. This is the only -root choice that is compatible with WebSphere Application Servers prior to v4.0 and with non-WebSphere Application Server name servers.

-url *some provider URL*
Indicates the value for the java.naming.provider.url property used to get the initial JNDI context. This option can be used in place of the -host, -port, and -root options. If the -url option is specified, the -host, -port, and -root options are ignored.

-factory *com.ibm.websphere.naming.WsnInitialContextFactory*
Indicates the initial context factory to be used to get the JNDI initial context. The value defaults to: *com.ibm.websphere.naming.WsnInitialContextFactory* The default value generally does not need to be changed.

-startAt *some/subcontext/in/the/tree*
Indicates the path from the bootstrap host's root context to the top level context where the dump should begin. The utility recursively dumps subcontexts below this point. It defaults to an empty string, that is, the bootstrap host root context.

-format{jndi | ins}

Option	Description
jndi	The default. Displays name components as atomic strings.
ins	Shows name components parsed per INS rules (id.kind).

-report {short | long}

Option	Description
short	The default. Dumps the binding name and bound object type. This output is also provided by JNDI Context.list().
long	Dumps the binding name, bound object type, local object type, and string representation of the local object (that is, the IORs, string values, and other values that are printed). For objects of user-defined classes to display correctly with the long report option, it may be necessary to add their containing directories to the list of directories searched. Set the environment variable **WAS_USER_DIRS**. The value can include one or more directories, as for example: **Platform** **Command** **UNIX** `WAS_USER_DIRS=/usr/classdir1:/usr/classdir2 export WAS_USER_DIRS` **Windows NT** set `WAS_USER_DIRS=c:\classdir1;d:\classdir2` All zip, jar, and class files in the specified directories can then be resolved by the class loader when running **dumpNameSpace**.

-traceString *"some.package.name.to.trace.*=all=enabled"*
: Represents the trace string with the same format as that generated by the servers. The output is sent to the file, `DumpNameSpaceTrace.out`.

Example: Invoking the name space dump utility

It is often helpful to view a dump of the name space to understand why a naming operation is failing. You can invoke the name space dump utility from the command line or from a program. Examples of each option follow.

Invoking name space dump utility from a command line

Invoke the name space dump utility from the command line by entering the following command:

```
dumpNameSpace -host myhost.mycompany.com -port 901
```

OR

```
dumpNameSpace -url corbaloc:iiop:myhost.mycompany.com:901
```

There are several command line options to choose from. For detailed help, enter the following command:

```
dumpNameSpace -help
```

For the z/OS environment: Use the `dumpNameSpace.sh` command. (Add `.sh` to the utility name.)

Invoking name space dump utility from a Java program

You can dump name spaces from a program with the com.ibm.websphere.naming.DumpNameSpace API. Refer to the WebSphere Application Server API documentation for details on the DumpNameSpace program interface

The following example illustrates how to invoke the name space dump utility from a Java program:

```
{
   ...
   import javax.naming.Context;
   import javax.naming.InitialContext;
   import com.ibm.websphere.naming.DumpNameSpace;
   ...
   java.io.PrintStream filePrintStream = ...
   Context ctx = new InitialContext();
   // Starting context for dump
   ctx = (Context) ctx.lookup("cell/nodes/node1/servers/server1");
   DumpNameSpace dumpUtil =
      new DumpNameSpace(filePrintStream, DumpNameSpace.SHORT);
   dumpUtil.generateDump(ctx);
   ...
}
```

Name space dump utility for`java:`, `local:` and `server` name spaces

Sometimes it is helpful to dump the java: name space for a J2EE application. You cannot use the dumpNameSpace command line utility for this purpose because the application's java: name space is accessible only by that J2EE application. From the WebSphere Application Server scripting tool, you can invoke a NameServer MBean to dump the java: name space for any J2EE application running in that same server process.

There is another name space local to server process which you cannot dump with the dumpNameSpace command line utility. This name space has the URL scheme of local: and is used by the container to bind objects locally instead of through the name server. The local: name space contains references to enterprise beans with local interfaces. There is only one local: name space in a server process. You can dump the local: name space by invoking the NameServer MBean associated with that server process.

Name space dump options

Name space dump options are specified in the MBean invocation as a parameter in chararacter string format. The option descriptions follow.

-startAt *some/subcontext/in/the/tree*
Indicates the path from the name space root context to the top level context where the dump should begin. The utility recursively dumps subcontexts below this point. It defaults to an empty string, that is, the root context.

-report {short | long}

Option	Description
short	The default. Dumps the binding name and bound object type. This output is also provided by JNDI Context.list().
long	Dumps the binding name, bound object type, local object type, and string representation of the local object (that is, the IORs, string values, and other values that are printed).

-root {tree | host | legacy | cell | node | server | default}
Specify the root context of where the dump should start. The default value for `-root` is *cell*. This option is only valid for server name space dumps.

Option	Description
tree	Dump the tree starting at the tree root context.
host	Dump the tree starting at the server host root context (synonymous with "node").
legacy	Dump the tree starting at the legacy root context.
cell	Dump the tree starting at the cell root context. This is the default option.
node	Dump the tree starting at the node root context (synonymous with "host").
server	Dump the tree starting at the server root context. This is `-root` default.
default	Dump the tree starting at the initial context which JNDI returns by default for that server type.

-format {jndi | ins}

Specify the format to display name component as atomic strings or parsed according to INS rules (id.kind). This option is only valid for server name space dumps.

Option	Description
jndi	Display name components as atomic strings. This is `-format` default.
ins	Display name components parsed according to INS rules (id.kind).

NameServer MBean invocation

1. Enter the WebSphere Application Server scripting command prompt.

   Invoke a method on a NameServer MBean by using the WebSphere Application Server scripting tool. Enter the scripting command prompt by typing the following command:

Platform	Command
UNIX	wsadmin.sh
Windows NT	wsadmin

   Use the -help option for help on using the wsadmin command.
2. Select the NameServer MBean instance to invoke.

   Execute the following script commands to select the NameServer instance you want to invoke. For example,

   ```
   set mbean [$AdminControl completeObjectName WebSphere:*,type=NameServer,cell=
       cellName,node=nodeName,process=serverName]
   ```

   where *cellName*, *nodeName*, and *serverName* are the names of the cell, node, and server for the MBean you want to invoke. The specified server must be running before you can invoke a method on the MBean.

   You can see a list of all NameServer MBeans current running by issuing the following query:

   ```
   $AdminControl queryNames {*:*,type=NameServer}
   ```
3. Invoke the NameServer MBean.

java: name space

Dump a java: name space by invoking the dumpJavaNameSpace method on the NameServer MBean. Since each server application has its own java: name space, the application must be specified on the method invocation. An application is identified by the application name, module name, and component name. The method syntax follows:

```
$AdminControl invoke $mbean dumpJavaNameSpace
  {{appname}{modName}{compName}{opts}}
```

where *appName* is the application name, *modName* is the module name, and *compName* is the component name of the java: name space you want to dump. The value for *opts* is the list of name space dump options described earlier in this section. The list can be empty.

local: name space

Dump a java: name space by invoking the dumpLocalNameSpace method on the NameServer MBean. Since there is only one local: name space in a server process, you have to specify the name space dump options only.

```
$AdminControl invoke $mbean dumpLocalNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Server name space

Dump a server name space by invoking the dumpServerNameSpace method on an application server's NameServer MBean. This provides an alternative way to dump the name space on an application server, much like the dumpNameSpace command line utility.

```
$AdminControl invoke $mbean dumpServerNameSpace {{opts}}
```

where *opts* is the list of name space dump options described earlier in this section. The list can be empty.

Name space dump output

Name space dump output is sent to the console. It is also written to the file `DumpNameSpace.log`, in the server's log directory.

Example: Invoking the name space dump utility for `java:` and`local:` name spaces

It is often helpful to view the dump of a `java:` or `local:` name space to understand why a naming operation is failing. The NameServer MBean running in the application's server process can be invoked from the WebSphere Application Server scripting tool to generate a dump of these name spaces. Examples of NameServer MBean calls to generate dumps of java: and local: name spaces follow.

Dumping a `java:` name space

Assume you want to dump the java: name space of an application component running in server server1 on node node1 of the cell MyCell. The application name is AcctApp in module AcctApp.war, and the component name is Acct Servlet. The following script commands generate a long format dump of the application's java: name space of that application:

```
set mbean
 [$AdminControl completeObjectName
   WebSphere:*,type=NameServer,cell=MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpJavaNameSpace {
  {AcctApp}{AcctApp.war}{Acct Servlet}{-report long}}
```

Dumping a `local:` name space

Assume you want to dump the local: name space for the server server1 on node node1 of cell MyCell. The following script commands will generate a short format dump of that server's local name space:

```
set mbean
 [$AdminControl completeObjectName WebSphere:*type=NameServer,cell=
  MyCell,node=node1,process=server1]
$AdminControl invoke $mbean dumpLocalNameSpace {{-report short}}
```

Name space dump sample output

Name space dump output looks like the following example, which is the **SHORT** dump format:

```
Getting the initial context
Getting the starting context

==============================================================================
Name Space Dump
   Provider URL: corbaloc:iiop:localhost:9810
   Context factory: com.ibm.websphere.naming.WsnInitialContextFactory
   Requested root context: cell
   Starting context: (top)=outpostNetwork
   Formatting rules: jndi
   Time of dump: Mon Sep 16 18:35:03 CDT 2002
==============================================================================


==============================================================================
Beginning of Name Space Dump
==============================================================================

    1 (top)
    2 (top)/domain                                    javax.naming.Context
    2    Linked to context: outpostNetwork
    3 (top)/cells                                     javax.naming.Context
    4 (top)/clusters                                  javax.naming.Context
    5 (top)/clusters/Cluster1                         javax.naming.Context
    6 (top)/cellname                                  java.lang.String
    7 (top)/cell                                      javax.naming.Context
    7    Linked to context: outpostNetwork
    8 (top)/deploymentManager                         javax.naming.Context
    8    Linked to URL: corbaloc::outpost:9809/NameServiceServerRoot
    9 (top)/nodes                                     javax.naming.Context
   10 (top)/nodes/will2                               javax.naming.Context
   11 (top)/nodes/will2/persistent                    javax.naming.Context
   12 (top)/nodes/will2/persistent/SomeObject         SomeClass
   13 (top)/nodes/will2/nodename                      java.lang.String
   14 (top)/nodes/will2/domain                        javax.naming.Context
   14    Linked to context: outpostNetwork
   15 (top)/nodes/will2/cell                          javax.naming.Context
   15    Linked to context: outpostNetwork
   16 (top)/nodes/will2/servers                       javax.naming.Context
   17 (top)/nodes/will2/servers/server1               javax.naming.Context
   18 (top)/nodes/will2/servers/will2                 javax.naming.Context
   19 (top)/nodes/will2/servers/member2               javax.naming.Context
   20 (top)/nodes/will2/node                          javax.naming.Context
   20    Linked to context: outpostNetwork/nodes/will2
   21 (top)/nodes/will2/nodeAgent                     javax.naming.Context
```

```
22 (top)/nodes/outpost                            javax.naming.Context
23 (top)/nodes/outpost/node                       javax.naming.Context
23    Linked to context: outpostNetwork/nodes/outpost
24 (top)/nodes/outpost/nodeAgent                  javax.naming.Context
24    Linked to URL: corbaloc::outpost:2809/NameServiceServerRoot
25 (top)/nodes/outpost/persistent                 javax.naming.Context
26 (top)/nodes/outpost/nodename                   java.lang.String
27 (top)/nodes/outpost/domain                     javax.naming.Context
27    Linked to context: outpostNetwork
28 (top)/nodes/outpost/servers                    javax.naming.Context
29 (top)/nodes/outpost/servers/server1            javax.naming.Context
30 (top)/nodes/outpost/servers/server1/url        javax.naming.Context
31 (top)/nodes/outpost/servers/server1/url/CatalogDAOSQLURL
31                                                java.net.URL
32 (top)/nodes/outpost/servers/server1/mail       javax.naming.Context
33 (top)/nodes/outpost/servers/server1/mail/PlantsByWebSphere
33                                                javax.mail.Session
34 (top)/nodes/outpost/servers/server1/TransactionFactory
34                                                com.ibm.ejs.jts.jts.ControlSet$LocalFactory
35 (top)/nodes/outpost/servers/server1/servername     java.lang.String
36 (top)/nodes/outpost/servers/server1/WSsamples      javax.naming.Context
37 (top)/nodes/outpost/servers/server1/WSsamples/TechSampDatasource
37                                                TechSamp
38 (top)/nodes/outpost/servers/server1/thisNode       javax.naming.Context
38    Linked to context: outpostNetwork/nodes/outpost
39 (top)/nodes/outpost/servers/server1/cell       javax.naming.Context
39    Linked to context: outpostNetwork
40 (top)/nodes/outpost/servers/server1/eis        javax.naming.Context
41 (top)/nodes/outpost/servers/server1/eis/DefaultDatasource_CMP
41                                                Default_CF
42 (top)/nodes/outpost/servers/server1/eis/WSsamples  javax.naming.Context
43 (top)/nodes/outpost/servers/server1/eis/WSsamples/TechSampDatasource_CMP
43                                                TechSamp_CF
44 (top)/nodes/outpost/servers/server1/eis/jdbc       javax.naming.Context
45 (top)/nodes/outpost/servers/server1/eis/jdbc/PlantsByWebSphereDataSource_CMP
45                                                PLANTSDB_CF
46 (top)/nodes/outpost/servers/server1/eis/jdbc/petstore
46                                                javax.naming.Context
47 (top)/nodes/outpost/servers/server1/eis/jdbc/petstore/PetStoreDB_CMP
47                                                PetStore_CF
48 (top)/nodes/outpost/servers/server1/eis/jdbc/CatalogDB_CMP
48                                                Catalog_CF
49 (top)/nodes/outpost/servers/server1/jta        javax.naming.Context
50 (top)/nodes/outpost/servers/server1/jta/usertransaction
50                                                java.lang.Object
51 (top)/nodes/outpost/servers/server1/DefaultDatasource
51                                                Default Datasource
52 (top)/nodes/outpost/servers/server1/jdbc       javax.naming.Context
53 (top)/nodes/outpost/servers/server1/jdbc/CatalogDB CatalogDB
54 (top)/nodes/outpost/servers/server1/jdbc/petstore  javax.naming.Context
55 (top)/nodes/outpost/servers/server1/jdbc/petstore/PetStoreDB
55                                                PetStoreDB
56 (top)/nodes/outpost/servers/server1/jdbc/PlantsByWebSphereDataSource
56                                                PLANTSDB
57 (top)/nodes/outpost/servers/outpost            javax.naming.Context
57    Linked to URL: corbaloc::outpost:2809/NameServiceServerRoot
58 (top)/nodes/outpost/servers/member1            javax.naming.Context
59 (top)/nodes/outpost/cell                       javax.naming.Context
59    Linked to context: outpostNetwork
60 (top)/nodes/outpostManager                     javax.naming.Context
61 (top)/nodes/outpostManager/domain              javax.naming.Context
61    Linked to context: outpostNetwork
62 (top)/nodes/outpostManager/cell                javax.naming.Context
62    Linked to context: outpostNetwork
63 (top)/nodes/outpostManager/servers             javax.naming.Context
64 (top)/nodes/outpostManager/servers/dmgr        javax.naming.Context
64    Linked to URL: corbaloc::outpost:9809/NameServiceServerRoot
```

```
  65 (top)/nodes/outpostManager/node                    javax.naming.Context
  65    Linked to context: outpostNetwork/nodes/outpostManager
  66 (top)/nodes/outpostManager/nodename                java.lang.String
  67 (top)/persistent                                   javax.naming.Context
  68 (top)/persistent/cell                              javax.naming.Context
  68    Linked to context: outpostNetwork
  69 (top)/legacyRoot                                   javax.naming.Context
  69    Linked to context: outpostNetwork/persistent
  70 (top)/persistent/AnotherObject                     AnotherClass

==============================================================================
End of Name Space Dump
==============================================================================
```

Naming and directories: Resources for learning

Use the following links to find relevant supplemental information about naming and directories. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

Programming instructions and examples
- Naming in WebSphere Application Server V5: Impact on Migration and Interoperability

Programming specifications
- Java Naming and Directory Interface™ 1.2.1 Specification
- OMG CosNaming Interoperable Naming Specification

Chapter 15. Using the dynamic cache service to improve performance

The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. For example, it intercepts calls through a servlet service() method or a command execute() method, and either stores the output of the object to, or serves the content of the object from the dynamic cache.

WebSphere Application Server, Version 4.0.1, supported the configuration of dynamic servlet caching through the use of a `servletcache.xml` file. For migration purposes, this file is still supported by this release. In order to utilize the new and improved functionality of the dynamic cache service in this release, you must configure your cache policy using the new `cachespec.xml` format.

The dynamic caching documentation provides you with the following tasks to enable and configure the dynamic cache service, as well as advanced features, such as controlling external caches and building user-defined drop-in components to customize the cache operation.

1. Enable the dynamic cache service globally.
2. Configure servlet caching.
3. Configure Edge Side Include (ESI) caching.
4. Configure command caching.
5. Configure Web services caching.
6. Troubleshoot any problems with the dynamic cache service.

Dynamic cache

Caching the output of servlets, commands and JavaServer Pages (JSP) files, improves application performance. WebSphere Application Server consolidates several caching activities, including servlets, Web services, and WebSphere commands into one service called the *dynamic cache*. These caching activities work together to improve application performance, and share many configuration parameters, which are set in the dynamic cache service of an application server.

You can use the dynamic cache to improve the performance of servlet and JSP files by serving requests from an in-memory cache. Cache entries contain servlet output, results of servlet execution, and metadata.

Configuring cache replication

Cache replication leverages the WebSphere internal replication service that is also leveraged for HttpSession memory-to-memory replication for failover purposes. Hence, a replication domain with at least one replicator entry needs to exist in order to replicate the data. The dynamic cache service, in essence, connects to the replicator. See more information in the topic referring to managing internal replication.

To configure cache replication and its features:

1. Click **Servers > Application Servers** in the administrative console navigation tree.

© Copyright IBM Corp. 2003

2. Click *server*.
3. Click **Dynamic Cache Service**.
4. Select the **Enable cache replication** check box in the **Cache replication** field.

   To manage **batch update** or **PUSH-PULL - PUSH/PULL**, repeat steps 1-4, then click the **Enable cache replication** link to the right of the **Enable cache replication** check box. Batch update interval is set under **push frequency**. PUSH-PULL-PUSH/PULL is set through the **runtime mode**.

   You can also select which replication domain and initial replicator entry the dynamic cache will utilize (either those managed within the cell or across the cell).

Cache replication

Data is generated one time and copied or replicated to other servers in the cluster, thus saving execution time and resources. Caching in a cluster has additional concerns. In particular, the same data could be required, and hence, generated in multiple places. Also, the access the resources need to generate the cached data can be restricted, preventing access to the data.

Cache replication addresses these concerns by generating the data one time and copying or replicating it to the other servers in the cluster. It also aids in cache consistency, in that cache entries that are not needed are removed or replaced.

The configuration specific to replication of data can exist as part of the Web container dynamic cache configuration accessible through the administrative console, or on a per cache entry basis through the `cachespec.xml` file. This includes the option to configure cache replication at the Web container level, but disabling it for a specific cache entry.

Cache replication can take on three forms:
- **PUSH** - Send out new entries, both ID and data, and updates to those entries.
- **PULL** - Requests data from other servers in the cluster when that data is not locally present. This mode of replication is not recommended.
- **PUSH/PULL** - Sends out IDs for new entries, then, only request from other servers in the cluster entries for IDs previously broadcast. The dynamic cache always sends out cache entry invalidations.

The dynamic cache provides a batch update option. Specifically, for PUSH or PUSH/PULL, the dynamic cache broadcasts the update asynchronously, based on a timed interval rather than sending them immediately upon inception. Invalidators are sent immediately. Distribution of invalidations addresses the issue of stale data residing in a cluster.

Internal messaging configuration settings

Use this page to set advanced configurations for Memory to Memory session replication.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service**> **Cache replication** > **Enable cache replication** .

The advanced replication settings include fields for choosing the initial replicator entry that connects to the replicator domains. As an alternative, you can specify the IP addresses and ports (of the form address:port) for connection to replicators outside of the cell that the server is administered under. By default, if a replicator is defined on the server you are configuring, that server is the one chosen for cache replication. Select the advanced properties only if you want to deviate from the default setting.

Note: The cache replication function can only be used if you are running the WebSphere Application Server for z/OS product on multiple OS images.

Internal messaging server

Specifies a domain from which your data will be replicated. Depending on the domain you choose to replicate the data, you can choose any of the replicators defined under that domain. You can use the default domain or choose one from the drop down window.

Runtime mode

Specifies the global sharing policy for this server.

The following settings are available:

- **Both push and pull** sends the cache ID of newly updated content to other servers in the replication domain. Then, if one of the other servers requests the content, and that server has the ID of the cache entry for the previously updated content, it will retrieve the content from the publishing server. On the other hand, if a request is made for an ID which has not been previously published, the server assumes it does not exist in the cluster and creates a new entry.
- **Push only** sends the cache ID and cache content of new content to all other servers in the replication domain.
- The sharing policy of **Not Shared** results in the cache ID and cache content not being shared with other servers in the replication domain.

The default setting for a non-clustered environment is **Not Shared**. When enabling replication, the default value is **Push only**.

Push frequency

Specifies the time in seconds to wait before pushing new or modified cache entries to other servers. A value of 0 (zero) means send immediately.

Setting this property to a value greater than 0 (zero) causes a ″batch″ push of all cache entries that are created or modified during the time period.

Default	0 (equivalent to immediate)

Enabling the dynamic cache service

In order to use the dynamic cache service, you must first enable it.

1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the administrative console navigation tree.
3. Click a server.
4. Click **Dynamic Cache Service** under Additional Properties.
5. Select **Enable service at server startup** in the **Startup state** field.
6. Click **Apply** or **OK**.
7. Restart WebSphere Application Server.

The dynamic cache service will now cache content for requests that have cache policies configured.

Dynamic cache service settings

Use this page to configure and manage the dynamic cache service settings.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service**.

Startup state

Specifies whether the dynamic cache is enabled.

Cache size

Specifies a positive integer as the value for the maximum number of entries the cache holds.

Enter the cache size value in this field between the range of 100 through 200,000.

Default priority

Specifies the default priority for cache entries, determining how long an entry stays in a full cache.

Default	1
Range	1 to 255

Disk offload

Specifies whether disk offload is enabled.

By default, the dynamic cache maintains the number of entries configured in memory. If new entries are created while the cache is full, the priorities configured for each cache entry and a least recently used algorithm, are used to remove entries from the cache. In addition to having a cache entry removed from memory when the cache is full, you can enable disk offload to have a cache entry copied to the file system (the location is configurable). Later, if that cache entry is needed, it is moved back to memory from the file system.

Cache replication

Specifies whether cache replication is enabled.

You can also configure advanced cache replication settings.

Configuring servlet caching

To enable servlet caching, you must enable the dynamic cache service.

1. Open the administrative console.
2. Click **Servers** > **Application Servers** in the console navigation tree.
3. Click a server.
4. Click **Web Container**.
5. Select the **Enable servlet caching** check box under the Configuration tab.
6. Click **Apply** or **OK**.

Servlet caching

After a servlet is invoked and generating the output to cache, a cache entry is created containing the output and the side effects of the invocation. For example, these side effects can include calls to other servlets or JavaServer Pages (JSP) files, as well as metadata about the entry, including timeout and entry priority information.

Unique entries are distinguished by an ID string generated from the HttpServletRequest object for each invocation of the servlet. You can then base servlet caching on:

- Request parameters and attributes the URI used to invoke the servlet
- Session information
- Other options, including cookies

Since JSP files are compiled by WebSphere Application Server into servlets, the dynamic cache function treats them the same, except in specifically documented situations.

Configuring the dynamic cache disk offload

By default, when the number of cache entries reaches the configured limit for a given WebSphere server, eviction of cache entries occurs, allowing new entries to enter the cache service. The dynamic cache includes an alternative feature named disk offload, that copies the evicted cache entries to disk for potential future access.

To configure disk offload:

1. Open the administrative server.
2. Click **Server** > **Application Server** in the administrative console navigation tree.
3. Click *server*.
4. Click **Dynamic Cache Service**.
5. Click the **Enable disk offload** check box in the **Disk offload** field. You can also set the disk offload location in this field.
6. Click **Apply** or **OK**.

Application servers must have different disk offload locations

When you have two or more application servers with servlet caching enabled and the application servers specify the same disk offload location for their caches through the dynamic cache service, the following exceptions might occur:

```
java.lang.NullPointerException
       at com.ibm.ws.cache.CacheOnDisk.readTemplate(CacheOnDisk.java:686)
       at com.ibm.ws.cache.Cache.internalInvalidateByTemplate(Cache.java:828)
```

or:

```
java.lang.NullPointerException
       at com.ibm.ws.cache.CacheOnDisk.readCacheEntry(CacheOnDisk.java:600)
       at com.ibm.ws.cache.Cache.getCacheEntry(Cache.java:341)
```

If one server is run as root and the other servers are run as nonroot, this problem could occur. For example, if `server1` runs as root and `server2` runs as `wasuser` or `wasgroup`, the cache files in the disk offload location might be created with root permissions. This situation causes the applications running on the nonroot servers to crash when they try to read or write to the cache.

The disk offload location must be unique for servers defined on the same node. If you have multiple servers defined on the same node, make sure the disk offload location is different for each server as defined on the **Dynamic Cache Service** panel, **Offload location** field.

Configuring Edge Side Include caching

Edge Side Include (ESI) is configured through the `plugin-cfg.xml` file.

The Web server plug-in contains a built-in ESI processor. The ESI processor has the ability to cache whole pages, as well as fragments, providing a higher cache hit

ratio. The cache implemented by the ESI processor is an in-memory cache, not a disk cache, therefore, the cache entries are not saved when the Web server is restarted.

The basic operation of the ESI processor is as follows: When a request is received by the Web server plug-in, it is sent to the ESI processor, unless the ESI processor is disabled. It is enabled by default. If a cache miss occurs, a Surrogate-Capabilities header is added to the request and the request is forwarded to the WebSphere Application Server. If the dynamic servlet cache is enabled in the application server, and the response is edge cacheable, the application server returns a Surrogate-Control header in response to the WebSphere Application Server plug-in.

The value of the Surrogate-Control response header contains the list of rules which are used by the ESI processor in order to generate the cache ID. The response is then stored in the ESI cache, using the cache ID as the key. For each ESI include tag in the body of the response, a new request is processed such that each nested include results in either a cache hit or another request forwarded to the application server. When all nested includes have been processed, the page is assembled and returned to the client.

The ESI processor is configurable through the WebSphere Web server plug-in configuration file `plugin-cfg.xml`. The following is an example of the beginning of this file, which illustrates the ESI configuration options.

```
<?xml version-"1.0"?>
<Config>
  <Property Name="esiEnable" Value="true"/>
  <Property Name="esiMaxCacheSize" Value="1024"/>
  <Property Name="esiInvalidationMonitor" Value="false"/>
```

The first option, esiEnable, can be used to disable the ESI processor by setting the value to false. ESI is enabled by default. If ESI is disabled, then the other ESI options are ignored.

The second option, esiMaxCacheSize, is the maximum size of the cache in 1K byte units. The default maximum size of the cache is 1 megabyte. If the cache is full, the first entry to be evicted from the cache is the entry that is closest to expiration.

The third option, esiInvalidationMonitor, specifies whether or not the ESI processor should receive invalidations from the application server. ESI works well when the Web servers following a threading model is used, and only one process is started. When multiple processes are started, each process caches the responses independently and the cache is not shared. This could lead to a situation where, the system's memory is fully used up by ESI processor.There are three methods by which entries are removed from the ESI cache: first, an entry's expiration timeout could fire; second, an entry may be purged to make room for newer entries; or third, the application server could send an explicit invalidation for a group of entries. In order for the third mechanism to be enabled, the esiInvalidationMonitor property must be set to true and the DynaCacheEsi application must be installed on the application server. The DynaCacheEsi application is located in the installableApps directory and is named DynaCacheEsi.ear. If the ESIInvalidationMonitor property is set to true but the DynaCacheEsi application is not installed, then errors will occur in the webserver plugin and the request will fail.

The ESI processor's cache can be monitored through the CacheMonitor application. In order for ESI processor's cache to be visible in the CacheMonitor, the

DynaCacheEsi application must be installed as described above and the ESIInvalidationMonitor property must be set to true in the `plugin-cfg.xml` file.

When WebSphere Application Server is used to serve static data, such as images and HTML on the application server, the URLs are also cached in the ESI processor. This data has a default timeout of 300 seconds. You can change the timeout value by adding the property com.ibm.servlet.file.esi.timeOut to your JVM's command line parameters. The following example shows how to set a one minute timeout on static data cached in the plug-in:

```
-Dcom.ibm.servlet.file.esi.timeOut=60
```

For more information about the `plugin-cfg.xml` file see ″Chapter 15, "Using the dynamic cache service to improve performance," on page 829.″

5.0.1 + For information about configuring alternate URL, see ″Configuring alternate URL.″

Configuring alternate URL

Alternate URL is a method for edge caching JavaServer Pages (JSP) files and servlet responses that you can not request externally. Dynamic cache provides support to recognize the presence of an Edge Side Include (ESI) processor and to generate ESI include tags and appropriate cache policies for edge cacheable fragments. However, for a fragment to be edge cacheable, you must be able to externally request it from the application server. In other words, if a user types the URL in their browser with the appropriate parameters and cookies for the fragment, WebSphere Application Server must return the content for that fragment.

One of the standard J2EE programming architectures is the model-view-controller (MVC) architecture, where a call to a controller servlet might include one or more child JSP files to construct the view. When using the MVC programming model, the child JSP files are edge cacheable only if you can request these JSP files externally, which is not usually the case. For example, if a child JSP file uses one or more request attributes that are determined and set by the controller servlet, you cannot cache that JSP file on the edge. You can use alternate URL support to overcome this limitation by providing an alternate controller servlet URL used to invoke the JSP file.

The alternate URL for a JSP file or a servlet is set in the `cachespec.xml` file as a property with the name `alternate_url`. You can set the alternate URL either on a per cache-entry basis or on a per cache-id basis. It is valid only if the `EdgeCacheable` property is also set for that entry. If the `EdgeCacheable` property is not set, the `alternate_url` property is ignored. The following is a sample cache policy using the `alternate_url` property:

```
<cache-entry>
    <class>servlet</class>
    <name>/AltUrlTest2.jsp</name>
    <property  name="EdgeCacheable">true</property>
    <property  name="alternate_url">/alturlcontroller2</property>
        <cache-id>
            <timeout>600</timeout>
            <priority>2</priority>
        </cache-id>
</cache-entry>
```

For more information on the `cachespec.xml` file, see Cachespec.xml file.

Configuring external cache groups

The dynamic cache can control caches outside of the application server, such as IBM Edge Server, an IBM HTTP Server for distributed platforms' Fast Response Cache Accelerator (FRCA) cache, and a WebSphere HTTP Server for distributed platforms plug-in ESI Fragment Processor. When external cache groups are defined, the dynamic cache matches externally cacheable cache entries with those groups, and pushes cache entries and invalidations out to those groups. This allows WebSphere Application Server to manage dynamic content beyond the application server. The content can then be served from the external cache, instead of the application server, improving savings in performance.

1. Open the administrative console.
2. Enable the dynamic cache.
   a. Click **Servers** > **Application Servers** in the administrative console navigation tree.
   b. Click a *server*.
   c. Click **Dynamic Cache Service**.
   d. Select the check box in the **Startup state** field to enable the dynamic cache.
3. Define the external cache group in which WebSphere Application Server should control.
   a. Click **External Caching Groups** from the Dynamic Cache administrative console page.
   b. Click **New** or choose an external cache group from the list.
4. Configure cache group members.
   a. Click **External cache groups** from the Dynamic Cache administrative console page. Then click **New** or choose an external cache group from the list.
   b. Click **External cache group members** > **New** or choose an external cache group member from the list.
   c. Type the configuration string in the **Address** field.
   d. Type the adapter bean name in the **Adapter Bean Name** field.
   e. **Save** the configuration.
   f. Click **Apply** or **OK**.

External cache group collection

Use this page to define sets of external caches controlled by WebSphere Application Server on Web servers, such as IBM Edge Server and IBM HTTP Server.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache Groups**.

Name:

Specifies the external cache group name.

The external cache group name needs to match the externalcache property as defined in the servlet or JSP `cachespec.xml` file.

When external caching is enabled, the cache matches pages with its URIs and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of the application server.

Type:

Specifies the external cache group type.

External cache group settings

Use this page to configure sets of external caches controlled by WebSphere Application Server on Web servers, such as IBM Edge Server and IBM HTTP Server.

To view this administrative console page, click **Servers** > **Application Server** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group*.

Name:

Specifies the external cache group name.

The external cache group name needs to match the externalcache property as defined in the servlet or JavaServer Pages (JSP) `cachespec.xml` file.

When external caching is enabled, the cache matches pages with its URIs and pushes matching pages to the external cache. The entries can then be served from the external cache, instead of the application server. This ability creates a significant savings in performance.

Type:

Specifies the external cache group type.

External cache group member collection

Use this page to define specific caches that are members of a cache group.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group* > **External cache group members**.

Address:

Specifies a configuration string used by external cache adapter bean to connect to the external cache.

AdapterBeanName:

Specifies the adapter bean name.

Example adapter bean names supported in WebSphere Application Server are:

AFPA
AdapterBeanName: com.ibm.ws.cache.servlet.Afpa
Address: Port on which afpa listens
ESI
AdapterBeanName: com.ibm.websphere.servlet.cache.ESIInvalidatorServlet
Address: local host
WTE

AdapterBeanName: com.ibm.websphere.edge.dynacache.WteAdapter
Address: hostname:port (host name and port on which WTE is listening)

External cache group member settings

Use this page to configure specific caches that are members of a cache group.

To view this administrative console page, click **Servers** > **Application Servers** > *server* > **Dynamic Cache Service** > **External Cache groups** > *external_cache_group* > **External cache group members** > *external_cache_group_member*.

Address:

Specifies a configuration string used by external cache adapter bean to connect to the external cache.

AdapterBeanName:

Specifies the adapter bean name.

Example adapter bean names supported in WebSphere Application Server are:

AFPA
AdapterBeanName: com.ibm.ws.cache.servlet.Afpa
Address: Port on which afpa listens
ESI
AdapterBeanName: com.ibm.websphere.servlet.cache.ESIInvalidatorServlet
Address: local host
WTE
AdapterBeanName: com.ibm.websphere.edge.dynacache.WteAdapter
Address: hostname:port (host name and port on which WTE is listening)

Configuring high-speed external caching through the Web server

IBM HTTP Server for Windows NT and Windows 2000 operating systems contains a high-speed cache referred to as the *Fast Response Cache Accelerator*, or *cache accelerator*.

The Fast Response Cache Accelerator is available on Windows NT and Windows 2000 operating systems and AIX platforms. However, support to cache dynamic content is only available on Windows NT and Windows 2000 operating systems.

You can enable cache accelerator to cache static and dynamic content. To enable cache accelerator for caching static content, add the following directives to the `http.conf` configuration file, in the IBM HTTP Server `conf` directory:

- `AfpaEnable`
- `AfpaCache on`
- `AfpaLogFile "`*`install_root`*`\IBMHttpServer\logs\afpalog" V-ECLF`

To enable cache accelerator for caching dynamic content, such as servlets and Java Server Pages (JSP) files, configure the WebSphere Application Server and the IBM HTTP Server for distributed platforms:

1. Configure WebSphere Application Server to enable Fast Response Cache Accelerator:

a. Configure an external cache group on the application server:
   - Click **Servers > Application Servers > server1**.
   - Click **Dynamic Cache Service** in the **Additional Properties** window.
   - Click **External Cache Groups** in the **Additional Properties** window.
   - Click **New** on the External cache group administrative console page to define an external cache group named `afpa` for each application server that uses the cache accelerator.
   - Type `afpa` in the External cache group field.
   - Click **Apply**.
   - Add a member to the group with an adapter bean name of `com.ibm.ws.cache.servlet.Afpa`:

   Click **Afpa > External cache group members**. Click **New** on the External cache group members administrative console page. Type `com.ibm.ws.cache.servlet.Afpa` in the AdapterBean name field. Enter an unused port number in the Address field.

b. Add a cache policy in the `cachespec.xml` file for the servlet or JSP file you want to cache. Add the following property to the cache policy:

```
<property name="ExternalCache">afpa</property>
```

It is important to follow all the steps for every application server in the cluster.

2. Enable cache accelerator on the IBM HTTP Server for distributed platforms:

a. Add the following directives to the end of the `httpd.conf` file:
   - `AfpaEnable`
   - `AfpaCache on`
   - `AfpaLogFile "`*`install_root`*`\IBMHttpServer\logs\afpalog" V-ECLF`
   - `LoadModule afpaplugin_module` *`install_root`*`/bin/afpaplugin.dll`
   - `AfpaPluginHost` *`WAS_Hostname:port`*, where *`WAS_Hostname`* is the host name of the application server and *`port`* is the port you specified in the Address field while configuring the external cache group member

The LoadModule directive loads the IBM HTTP Server plug-in that connects the Fast Response Cache Accelerator to the WebSphere Application Server fragment cache. If multiple IBM HTTP Servers are routing requests to a single application server, add the directives above to the `http.conf` file of each of these IBM HTTP Servers for distributed platforms. If one IBM HTTP Server is routing requests to a cluster of application servers, add the `AfpaPluginHost WAS_Hostname:port` directive to the `http.conf` file for each application server in the cluster. For example, if there are three application servers in the cluster, add the following directives to the `http.conf` file:
- `LoadModule afpaplugin_module` *`install_root`*`/bin/afpaplugin.dll`
- `AfpaPluginHost WAS1_Hostname:port1`
- `AfpaPluginHost WAS2_Hostname:port2`
- `AfpaPluginHost WAS3_Hostname:port3`

Configuring Fast Response Cache Accelerator cache size through a distributed platforms Web server:

In the default IBM HTTP Server for distributed platforms configuration, the maximum Fast Cache Accelerator dynamic cache size is calculated as 1/8 of physical pin-able memory. On a machine with 384 megabytes of RAM, it allows a maximum of approximately 50 megabytes for the Fast Cache Accelerator dynamic cache. When this limit is reached, the Cache Accelerator then deletes older entries to cache new ones.

Follow these steps to configure the Cache Accelerator:

Using the IBM HTTP Server for distributed platforms' AfpaDynaCacheMax directive, tune the maximum allowed cache size:

1. Place the directive in the global server configuration scope, along with the other default Fast Cache Accelerator directives.
2. Enable Fast Cache Accelerator. To enable the Fast Cache Accelerator, update the following directives in this IBM HTTP Server's `http.conf` file:

   ```
   AfpaEnable
   AfpaCache on
   AfpaLogFile "c:/Program Files/IBM HTTP Server/logs/afpalog" V-ECLF
   AfpaDynaCacheMax 10
   ```

   These above settings limit the dynamic cache size to 10 megabytes. If you use these directives to increase cache size, do not make the cache so large that all the physical memory is consumed. Determine how much memory is available when all applications are running, by using the Windows Task Manager.

   Assign no more than 50% of available physical memory to the dynamic cache. Specifying too large a cache not only decreases the performance of other applications, but also puts you at a risk for completely running out of memory.

   The default configuration does not include the AfpaDynaCacheMax directive where the cache size is automatically calculated as 1/8 of physical memory.

Displaying cache information

The dynamic cache monitor is an installable Web application that displays simple cache statistics, cache entries, and cache policy information.

The cache monitor provides information on the cache in the Servant Region to which your browser connects to interact with the monitor. Thus, in an environment with multiple Servant Regions, the cache monitor provides a partial view of caching activity.

1. Use the administrative console to install the cache monitor application from the `*install_root*/installableApps` directory. The application is named `CacheMonitor.ear`. Install the cache monitor onto the application server you are trying to monitor. Installing the cache monitor on the *admin_host* (port 909*x*) is more secure than installing it on the *default_host* (908*x*), and so it is preferable to install it onto the *admin_host*.
2. Access the Web application using a Web browser and the URL `http://`*`your host_name`*`:`*`your port_number`*`/cachemonitor`, where *your port_number* is the port associated with the host on which you installed the cache monitor application.
3. Verify that the cache monitor is working properly.
   a. View the Statistics page and verify the cache configuration and cache data. Click **Reset Statistics** to reset the counters
   b. View the Cache Policies page to see which cache policies are currently loaded in the dynamic cache. Click on a template to view the cache ID rules for the template.
   c. View the Cache Contents page to examine the contents currently cached in memory.
   d. View the ESI Statistics page to view data about the current ESI processors configured for caching. Click **Refresh Statistics** to see the latest statistics or content from the ESI processors. Click **Reset Statistics** to reset the counters.

e. View the Disk Offload page to view content currently off-loaded from memory to disk.

When viewing contents on memory or disk, click on a template to view all entries for that template, click on a dependency ID to view all entries for the ID, or click on the cache ID to view the entire data cached for that entry.

4. Use the cache monitor to perform basic operations on data in a cache.

Remove an entry from cache
Click **Invalidate** when viewing a cache entry.

Remove all entries for a certain dependency ID
Click **Invalidate** when viewing entries for a dependency ID.

Remove all entries for a certain template
Click **Invalidate** when viewing entries for a template.

Move an entry to the front of the Least Recently Used queue to avoid eviction
Click **Refresh** when viewing a cache entry.

Move an entry from disk to cache
Click **Send to Memory** when viewing a cache entry on disk.

Clear the entire contents of the cache
Click **Clear Cache** while viewing statistics or contents.

Clear the contents on the ESI processors
Click **Clear Cache** while viewing ESI statistics or contents.

Clear the contents of the disk cache
Click **Clear Disk** while viewing disk contents.

Configuring cacheable objects with the cachespec.xml file

Define cacheable objects inside the `cachespec.xml`, found inside the Web module `WEB-INF` or enterprise bean `META-INF` directory.

You can place a global `cachespec.xml` in the application server properties directory, but the recommended method is to place the cache configuration file with the deployment module. The root element of the `cachespec.xml` file is *<cache>*, which contains *<cache-entry>* elements.

Within a <cache-entry>...</cache-entry> element are parameters that allow you to complete the following tasks to enable the dynamic cache with the `cachespec.xml` file:

1. Develop a `cachespec.xml` file.
   a. Create a caching configuration file.

      In the `<install_root>/properties` directory, locate the `cachespec.sample.xml` file.
   b. Copy the `cachespec.sample.xml` file to `cachespec.xml` in Web module `WEB-INF` or enterprise bean `META-INF` directory.
2. Define the cache-entry elements necessary to identify the cacheable objects. See the topic "Cachespec.xml file" for a list of elements.
3. Develop cache-ID rules.

   To cache an object, WebSphere Application Server must know how to generate unique IDs for different invocations of that object. The <cache-id> element performs that task. Each cache entry can have multiple cache-ID rules that execute in order until either a rule returns non-empty cache-ID or no more rules remain to execute. If none of the cache-ID generation rules produce a valid cache ID, then the object is not cached. Develop these IDs in one of two ways:

- Use the <component> element defined in the cache policy of a cache entry (recommended)
- Write custom Java code to build the ID from input variables and system state

To configure the cache entry to use the IdGenerator, specify your IdGenerator in the XML file by using the `<idgenerator>` tag, for example:

```
<cache-entry>
     <class>servlet</class>
     <name>/servlet/CommandProcessor</name>
<cache-id>
     <idgenerator>com.mycompany.SampleIdGeneratorImpl</idgenerator>
     <timeout>60</timeout>
</cache-id>
</cache-entry>
```

5.0.1 **5.0.2** You can also use the Application Assembly Tool (AAT) to define the IdGenerator class in the cache policy's **Advanced** tab.

4. Specifying dependency ID rules. Use dependency ID elements to specify additional cache group identifiers that associate multiple cache entries to the same group identifier.

   The dependency ID is generated by concatenating the dependency ID base string with the values returned by its component elements. If a required component returns a null value, then the entire dependency ID does not generate and is not used. You can validate the dependency IDs explicitly through the WebSphere Dynamic Cache API, or use another cache-entry <invalidation> element. Multiple dependency ID rules can exist per cache-entry. All dependency ID rules separately execute. See the topic ″Cachespec.xml file″ for a list of <component> elements.
5. Invalidate other cache entries as a side effect of this object execution, if relevant. You can define invalidation rules in exactly the same manner as dependency IDs. However, the IDs that generate by invalidation rules are used to invalidate cache entries that have those same dependency IDs.

   The invalidation ID is generated by concatenating the invalidation ID base string with the values returned by its component element. If a required component returns a null value, then the entire invalidation ID is not generated and no invalidation occurs. Multiple invalidation rules can exist per cache-entry. All invalidation rules separately execute.
6. Verify the cacheable page.

Typically you declare several <cache-entry>...</cache-entry> elements inside a `cachespec.xml` file.

The dynamic cache responds to changes in this file. When new versions of the `cachespec.xml` are detected, the old policies are replaced. Objects cached through the old policy file are not automatically invalidated from the cache; they are either reused with the new policy or eliminated from the cache through its replacement algorithm.

For each of the three IDs (cache, dependency, invalidation) generated by cache entries, a <cache-entry> can contain multiple elements. The dynamic cache will execute the <cache-id> rules in order, and the first one that successfully generates an ID will be used to cache that output. If the object is to be cached, each one of the <dependency-id> elements will be executed to build a set of dependency IDs

for that cache entry. Finally, each of the <invalidation> elements will be executed, building a list of IDs that the dynamic cache will invalidate, whether or not this object is cached.

Verifying the cacheable page

Verify the cacheable page by following these steps:

1. View the snoop servlet in the default application by accessing the URI: `/snoop`
2. Invoke and reload the URI several times using a different Web browser or using different parameters. This action returns the same output for the snoop servlet. The snoop servlet is now operating incorrectly, because it displays the request information from its first invocation rather than from the current request.
3. Inspect the entry in the cache with the dynamic cache monitor.

Cachespec.xml file

The cache parses the `cachespec.xml` file on startup, and extracts from each <cache-entry> element a set of configuration parameters. Every time a new servlet or other cacheable object initializes, the cache attempts to match each of the different cache-entry elements, to find the configuration information for that object. Different cacheable objects have different <class> elements. You can define the specific object a cache policy refers to using the <name> element.

Location

The `cachespec.xml` file is found inside the `WEB-INF` directory of a Web module.

You can place a global `cachespec.xml` file in the application server properties directory, but the recommended method is to place the cache configuration file with the deployment module. (To place the cache configuration file with the deployment module, use the Assembly ToolkitApplication Assembly Tool (AAT) to define the cacheable objects.

The root element of the `cachespec.xml` file is *`cache`*, which contains *`cache-entry`* elements.

The `cachespec.dtd` file is available in the application server properties directory.

Usage notes

Each cache entry must specify certain basic information that the dynamic cache uses to process that entry. This section explains the function of each cache entry element of the `cachespec.xml` file including:

- class
- name
- sharing-policy
- property
- cache-id

class

```
<class>command | servlet | webservice</class>
```

This element is required and governs how the application server interprets the remaining cache policy definition. The value `servlet` refers to servlets and JavaServer Pages (JSP) files deployed in the WebSphere Application Server servlet engine. The `object` class extends the servlet with special component types for Web services requests. Finally, the value `command` refers to classes using the WebSphere command programming model. The following examples illustrate the `class` element:

```
<class>command</class>
<class>servlet</class>
<class>webservice</class>
```

name

```
<name>name</name>
```

where *name* is the fully qualified class name of the command, servlet, or object.

There are two ways to use <name> to specify a cacheable object:
- For commands and objects, this required element must include the package name, if any, and class name, including a trailing `.class`, of the configured object.
- For servlets and JSP files, if the `cachespec.xml` file is in the WebSphere Application Server properties directory, this required element must include the full URI of the JSP file or servlet to cache. For servlets and JSP files, if the `cachespec.xml` file is in the Web application, this required element can be relative to the specific Web application context root.

  Note: The preferred location of the `cachespec.xml` file is in the Web application, not the properties directory.

You can specify multiple <name> elements within a <cache-entry> if you have different mappings that refer to the same servlet.

The following examples illustrate the `name` element:

```
<name>com.mycompany.MyCommand.class</name>
<name>default_host:/servlet/snoop</name>
<name>com.mycompany.beans.MyJavaBean</name>
<name>mywebapp/myjsp.jsp</name>
```

sharing-policy

```
<sharing-policy> not-shared | shared-push | shared-pull </sharing-policy>
```

When working within a cluster with a distributed cache, these values determine the sharing characteristics of entries created from this object. If this element is not present, a not-shared value is assumed. Also, in non-distributed environments, not-shared is the only valid value. This property does not affect distribution to Edge servers through the Edge fragment caching property.

Value	Description
not-shared	Cache entries for this object are not shared among different application servers. These entries can contain non-serializable data. For example, a cached servlet can place non-serializable objects into the request attributes, if the <class> type supports it.

shared-push	Cache entries for this object are automatically distributed to the dynamic caches in other application servers or cooperating Java virtual machines (JVMs). Each cache has a copy of the entry at the time it is created. These entries cannot store non-serializable data.
shared-pull	Cache entries for this object are shared between application servers on demand. If an application server gets a cache miss for this object, it queries the cooperating application servers to see if they have the object. If no application server has a cached copy of the object, the original application server executes the request and generates the object. These entries cannot store non-serializable data. This mode of sharing is not recommended.
shared push-pull	Cache entries for this object are shared between application servers on demand. When an application server generates a cache entry, it broadcasts the cache ID of the created entry to all cooperating application servers. Each server then knows whether an entry exists for any given cache ID. On a given request for that entry, the application server knows whether to generate the entry or pull it from somewhere else. These entries cannot store non-serializable data.

The following example shows a sharing policy:

```
<sharing-policy>not-shared</sharing-policy>
```

property

```
<property name="key">value</property>
```

where *key* is the name of the property defined for this cache entry element, and *value* is the corresponding value.

You can set optional properties on a cacheable object, such as a description of the configured servlet. The class determines valid properties of the cache entry. At this time, the following properties are defined:

Property	Valid classes	Value
ApplicationName	All	Overrides the J2EEName application ID so that multiple applications can share a common cache ID namespace.

EdgeCacheable	Servlet	True or false. Default is false. If the property is true, then the given servlet or JSP file is externally requested from an Edge Server. Whether or not the servlet or JSP file is cacheable, depends on the rest of the cache specification.
ExternalCache	Servlet	Specifies the external cache name. The external cache name needs to match the external cache group name.
consume-subfragments	Servlet or Web service	True or false. Default is false. When a servlet is cached, only the content of that servlet is stored, and includes placeholders for any other fragments to which it includes or forwards. Consume-subfragments (CSF) tells the cache not to stop saving content when it includes a child servlet. The parent entry, the one marked CSF, includes all the content from all fragments in its cache entry, resulting in one big cache entry that has no includes or forwards, but the content from the whole tree of entries. This can save a significant amount of application server processing, but is typically only useful when the external HTTP request contains all the information needed to determine the entire tree of included fragments.
alternate_url	Servlet	Specifies the alternate URL used to invoke the servlet or JSP file. The property is valid only if the EdgeCacheable property also is set for the cache entry.
persist-to-disk	All	True or false. Default is true. When this property is set to false, the cache entry is not written to the disk when overflow or server stopping occurs.
save-attributes	Servlet	True or false. Default is true. When this property is set to false, the request attributes are not saved with the cache entry.

cache-id

To cache an object, the application server must know how to generate a unique ID for different invocations of that object. These IDs are built either from user-written custom Java code or from rules defined in the cache policy of each cache entry. Each cache entry can have multiple cache ID rules that are executed in order until either:
- A rule returns a non-empty cache ID, or
- No more rules are left to execute.

If none of the cache ID generation rules produce a valid cache ID, the object is not cached.

Each `cache-id` element defines a rule for caching an object and is composed of the sub-elements component, timeout, priority, and property. The following example illustrates a cache-id:

```
<cache-id>component*| timeout? | priority? | property* </cache-id>
```

component sub-element

Use the component sub-element to generate a portion of the cache ID. Each component sub-element consists of the attributes `id`, `type`, and `ignore-value`, and the elements `method`, `field`, `required`, `value`, and `not-value`.
- Use the `id` attribute to identify the component.
- Use the `type` attribute to identify the type of component. The following table lists the values for the type.

Type	Valid classes	Meaning
method	command	Calls the indicated method on the command or object
field	command	Retrieves the named field in the command or object
parameter	servlet	Retrieves the named parameter value from the request object
parameter-list	servlet	Retrieves a list of values for the named parameter
session	servlet	Retrieves the named value from the HTTPSession
cookie	servlet	Retrieves the named cookie value
attribute	servlet	Retrieves the named request attribute
header	servlet and Web service	Retrieves the named request header
pathInfo	servlet	Retrieves the pathInfo from the request
servletpath	servlet	Retrieves the servlet path
locale	servlet	Retrieves the request locale

SOAPEnvelope	Web service	Retrieves the SOAPEnvelope from a Web services request. An ID attribute of `Hash` uses a Hash of the SOAPEnvelope, while `Literal` uses the SOAPEnvelope as received.
SOAPAction	Web service	Retrieves the SOAPAction header, (if available), for a Web services request.
serviceOperation	Web service	Retrieves the service operation for a Web services request
serviceOperationParameter	Web service	Retrieves the specified parameter from a Web services request

- Use the `ignore-value` attribute to specify whether or not to use the value returned by this component in cache ID formation. This is an optional attribute with a default value of false. If the value is true, only the ID of the component is used when creating a cache ID, or no output is used when creating a dependency or invalidation ID.
- Use the **method** element to call a void method on a returned object. You can infinitely nest method and field objects in any combination. The method must be public and is not valid for edge-cacheable components. For example:

  ```
  <component id="getUser" type="method"><method>getUserInfo
  <method>getName</method></method></component>
  ```

  This method is equivalent to getUser().getUserInfo().getName()
- Use the **field** element to access a field in a returned object. You can infinitely nest method and field objects in any combination. The field must be public. Not valid for edge-cacheable components. For example:

  ```
  <component id="getUser" type="method"><method>getUserInfo
  <field>name</field></method></component>
  ```

  This method is equivalent to getUser().getUserInfo().name
- Use the **required** element to specify whether or not this component must return a non-null value for this cache ID for it to represent a valid cache. If set to `true`, this component must return a non-null value for this cache ID to represent a valid cache ID. If set to `false`, the default, a non-null value is used in the formation of the cache ID and a null value means that this component is not used at all in the ID formation. For example:

  ```
  <required>true</required>
  ```
- Use the **value** element to specify values that must match to use this component in cache ID formation. For example:

  ```
  <component id="getUser" type="method"><value>blue</value>
  <value>red</value> </component>
  ```
- Use the **not-value** element to specify values that must not match to use this component in cache ID formation. This method is similar to `<value>`, but instead prescribes the defined values from caching. You can use multiple `<not-value>` elements when there is more than one invalid value. For example:

  ```
  <component id="getUser" type="method">
  <required>true</required>
  <not-value>blue</not-value>
  <not-value>red</not-value></component>
  ```

The component element can have either a `method` or a `field` element, or a `value` or a `not-value` element. The `method` and `field` elements apply only to commands. The following example illustrates the attributes of a component element:

```
<component id="isValid" type="method" ignore-value="true"><component>
```

timeout sub-element

The timeout sub-element is used to specify a time-to-live (TTL) value for the cache entry. For example,

```
<timeout>value</timeout>
```

where *value* is the amount of time, in seconds, to keep the cache entry. If 0, or a negative value is specified, the cache entry is kept indefinitely.

priority sub-element

Use the priority sub-element to specify the priority of a cache entry in a cache. The priority weighting is used by the least recently used (LRU) algorithm, of the cache to decide which entries to remove from the cache if the cache runs out of storage space. For example,

```
<priority>value</priority>
```

where *value* is a positive integer between 1 and 255 inclusive.

property sub-element

Use the property sub-element to specify generic properties for the cache entry. For example,

```
<property name="key">value</property>
```

where *key* is the name of the property to define, and *value* is the corresponding value.

For example:

```
<property name="description">The Snoop Servlet</property>
```

Property	Valid classes	Meaning
sharing-policy/timeout/priority	All	Overrides the settings for the containing cache entry when the request matches this cache ID.
EdgeCacheable	servlet	Overrides the settings for the containing cache entry when the request matches this cache ID.

idgenerator and metadatagenerator elements

Use the `idgenerator` element to specify the class name loaded for the generation of the cache ID. The IdGenerator must implement the `com.ibm.websphere.servlet.cache.IdGenerator` interface. The IdGenerator must build and set cache IDs, group IDs and invalidation IDs. An example of the `idgenerator` element follows:

```
<idgenerator> classname classname </idgenerator>
```

(where classname= Fully qualified name of the class to use)

Use the `metadatagenerator` element to specify the class name loaded for the metadata generation cache ID. The MetadataGenerator class must implement the `com.ibm.websphere.servlet.cache.MetaDataGenerator` interface. The MetadataGenerator defines properties like timeout, external caching or generic properties. An example of the `metadatagenerator` element follows:

```
<metadatagenerator> classname classname </metadatagenerator>
```

(where classname= Fully qualified name of the class to use)

Configuring command caching

Cacheable commands are stored in the cache for re-use with a similar mechanism for servlets and Java Server Pages (JSP) files. However, in this case, the unique cache IDs are generated based on methods and fields present in the command as input parameters. For example, a **GetStockQuote** command can have a symbol as its input parameter.

A unique cache ID can generate from the name of the command, plus the value of the symbol.

To use command caching you must:
Create a command.

1. Define an interface. The Command interface specifies the most basic aspects of a command.

   You must define the interface that extends one or more of the interfaces in the command package. The command package consists of three interfaces:
   - TargetableCommand
   - CompensableCommand
   - CacheableCommand

   In practice, most commands implement the TargetableCommand interface, which allows the command to execute remotely. The code structure of a command interface for a targetable command follows:

   ```
   ...
   import com.ibm.websphere.command.*;
   public interface MyCommand extends TargetableCommand {
         // Declare application methods here
   }
   ```

1. Provide an implementation class for the interface. Write an interface that extends the CacheableCommandImpl class and implements your command interface. This class contains the code for the methods in your interface, the methods inherited from extended interfaces like the CacheableCommand interface, and the required or abstract methods in the CacheableCommandImpl class.

   You can also override the default implementations of other methods provided in the CacheableCommandImpl class.

Command class

To write a command interface, extend one or more of the three interfaces included in the command package. The base interface for all commands is the Command interface. This interface provides only the client-side interface for generic commands and declares three basic methods:

- **isReadyToCallExecute.** This method is called on the client side before the command passes to the server for execution.
- **execute.** This method passes the command to the target and returns any data.
- **reset.** This method reverts any output properties to the values they had before the execute method was called so that you can reuse the object.

The implementation class for your interface must contain implementations for the isReadyToCallExecute and reset methods.

CacheableCommandImpl class

Commands are implemented by extending the class CacheableCommandImpl, which implements the CacheableCommand interface.

The CacheableCommandImpl class is an abstract class that provides implementations for some of the methods in the CacheableCommand interface, for example, setting return values. This class declares additional methods that the application must implement, for example, how to execute the command.

The code structure of an implementation class for the CacheableCommand interface follows:

```
...
import com.ibm.websphere.command.*;
public class MyCommandImpl extends CacheableCommandImpl
implements MyCommand {
// Set instance variables here      ...
// Implement methods in the MyCommand interface      ...
// Implement abstract methods in the CacheableCommandImpl class
...
}
```

Example: Caching a command object

This example of command caching is a simple stock quote command.

The following is a stock quote command bean. It accepts a ticker as an input parameter and produces a price as its output parameter.

```
public class QuoteCommand extends CacheableCommandImpl
{
    private String ticker;
    private double price;
    // called to validate that command input parameters have been set
    public boolean isReadyToCallExecute() {
      return (ticker!=null);
    }
    // called by a cache-hit to copy output properties to this object
    public void setOutputProperties(TargetableCommand fromCommand) {
        QuoteCommand f = (QuoteCommand)fromCommand;
        this.price = f.price;
    }

   // business logic method called when the stock price must be retrieved
    public void performExecute()throws Exception {...}

    //input parameters for the command
    public void setTicker(String ticker) { this.ticker=ticker;}
    public String getTicker() { return ticker;}

    //output parameters for the command
    public double getPrice()  { return price;};
}
```

To cache the above command object using the stock ticker as the cache key and using a 60 second time-to-live, use the following cache policy:

```
<cache>
 <cache-entry>
  <class>command</class>
  <sharing-policy>not-shared</sharing-policy>
  <name>QuoteCommand</name>
   <cache-id>
    <component type="method" id="getTicker">
     <required>true</required>
    </component>
    <priority>3</priority>
    <timeout>60</timeout>
   </cache-id>
 </cache-entry>
</cache>
```

Example: Caching Web services

The following is a example of building a set of cache policies for a simple Web services application. The application in this example stores stock quotes, and has operations to read, update the price of, and buy a given stock symbol.

Following are two SOAP message examples that the application can receive, with accompanying HTTP Request headers.

The first message sample contains a SOAP message for a GetQuote operation, requesting a quote for IBM. This is a read-only operation that gets its data from the back-end, and is very cacheable. In this example the SOAP messasge is cached and a timeout is placed on its entries to guarantee the quotes it returns are not too out of date.

Message example 1

```
POST /soap/servlet/soaprouter
HTTP/1.1
Host: www.myhost.com
Content-Type: text/xml; charset="utf-8"
SOAPAction: urn:stockquote-lookup
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:getQuote xmlns:m="urn:stockquote:>
<symbol>IBM</symbol>
</m:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAPAction HTTP header in the request is defined in the SOAP specification and is used by HTTP proxy servers to dispatch requests to particular HTTP servers. WebSphere Application Server dynamic cache can use this header in its cache policies to build IDs without having to parse the SOAP message.

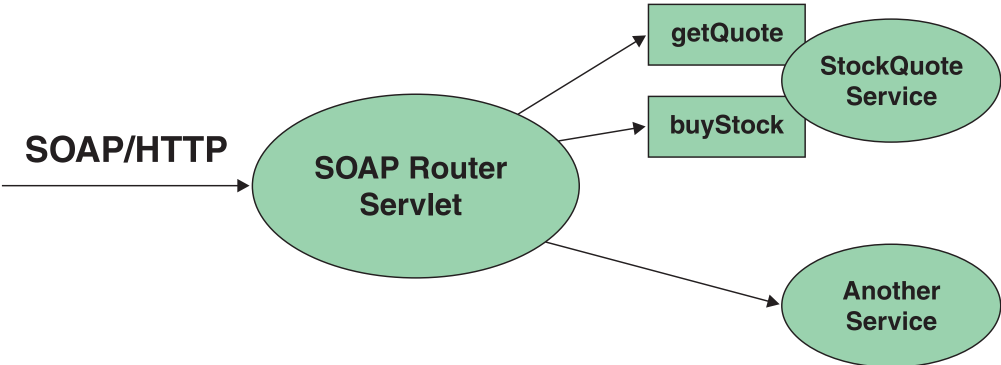
Message example 2 illustrates a SOAP message for a BuyQuote operation. While message 1 is cacheable, this message is not, because it updates the back-end database.

Message example 2

```
POST /soap/servlet/soaprouter
HTTP/1.1
Host: www.myhost.com
```

```
Content-Type: text/xml; charset="utf-8"
SOAPAction: urn:stockquote-update
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:buyStock xmlns:m="urn:stockquote:>
<symbol>IBM</symbol>
</m:getQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The graphic illustrates how to invoke methods with the SOAP messages. In Web services terms, especially Web Service Definition Language (WSDL), a service is a collection of operations such as getQuote and buyStock. A body element namespace (urn:stockquote in our example) defines a service, and the name of the first body element indicates the operation.



The following is an example of WSDL for the getQuote operation:

```
<?xml version="1.0"?>
<definitions name="StockQuoteService-interface"
targetNamespace="http://www.getquote.com/StockQuoteService-interface"
xmlns:tns="http://www.getquote.com/StockQuoteService-interface"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns=soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
<message name="SymbolRequest">
<part name="return" type="xsd:string"/>
</message>
<portType name="StockQuoteService">
<operation name="getQuote">
<input message="tns:SymbolRequest"/>
<output message="tns:QuoteResponse"/>
</operation>
</portType>
<binding name="StockQuoteServiceBinding"
type="tns:StockQuoteService">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="getQuote">
<soap:operation soapAction="urn:stockquote-lookup"/>
<input>
<soap:body use="encoded" namespace="urn:stockquote"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</input>
<output>
<soap:body use="encoded" namespace="urn:stockquotes"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
</output>
</operation>>
</binding>
</definition>
```

To build a set of cache policies for a Web services application configure WebSphere Application Server dynamic cache to recognize cacheable service operation of the operation.

WebSphere Application Server inspects the HTTP request to determine whether or not an incoming message can be cached based on the cache policies defined for an application. In this example, buyStock and stock-update are not cached, but stockquote-lookup is cached. In the `cachespec.xml` file for this Web application, the cache policies need defining for these services so that the dynamic cache can handle both SOAPAction and service operation.

WebSphere Application Server uses the operation and the message body in Web services cache IDs, each of which has a component associated with them. Therefore, each Web services `<cache-id>` rule contains only two components. The first is for the operation. Because you can perform the stockquote-lookup operation by either using a SOAPAction header or a service operation in the body, you must define two different `<cache-id>` elements, one for each method. The second component is of type ″body″, and defines how WebSphere Application Server should incorporate the message body into the cache ID. You can use a hash of the body, although it is legal to use the literal incoming message in the ID.

The incoming HTTP request is analyzed by WebSphere Application Server to determine which of the `<cache-id>` rules match. Then, the rules are applied to form cache or invalidation IDs.

The following is sample code of a `cachespec.xml` file defining SOAPAction and servicesOperation rules:

```
<cache>
<cache-entry>
<class>webservice</class>
<name>/soap/servlet/soaprouter</name>
<sharing-policy>not-shared</sharing-policy>
<cache-id>
<component id="" type=SOAPAction>
<value>urn:stockquote-lookup</value>
</component>
<component id="Hash" type="SOAPEnvelope"/>
<timeout>3600</timeout>
<priority>1<priority>
</cache-id>
<cache-id>
<component id="" type="serviceOperation">
<value>urn:stockquote:getQuote</value>
</component>
<component id="Hash" type="SOAPEnvelope"/>
<timeout>3600</timeout>
<priority>1</priority>
</cache-id>
</cache-entry>
</cache>
```

Example: Configuring the dynamic cache

This example puts all the steps together for configuring the dynamic cache with the `cachespec.xml` file, showing the use of the cache ID generation rules, dependency IDs, and invalidation rules.

Suppose we have a servlet which is used to manage a simple news site. This servlet uses the query parameter ″action″ to determine whether the request is

being used to ″view″ news or ″update″ news (used by the administrator). Further, another query parameter ″category″ is used to select the news category. Further, suppose that this site supports an optional customized layout, which is stored in the user's session using the attribute name ″layout″. Here are example URL requests to this servlet:

http://*yourhost/yourwebapp*/newscontroller?action=view&category=sports (Returns a news page for the sports category)

http://*yourhost/yourwebapp*/newscontroller?action=view&category=money (Returns a news page for the money category)

http://*yourhost/yourwebapp*/newscontroller?action=update&category=fashion (Allows the administrator to update news in the fashion category)

Here are the steps for configuring dynamic cache with `cachespec.xml`, using the information provided to you:

1. Define the cache-entry elements necessary to identify the servlet. In this case, the servlet's URI is ″newscontroller″ so this will be our cache-entry's name element. Also, since we are caching a servlet/JavaServer Page (JSP), the cache-entry class is ″servlet″.

   ```
   <cache-entry>
   <name> /newscontroller </name>
   <class>servlet  </class>
    </cache-entry>
   ```

2. Define cache ID generation rules. For this servlet, we only want to cache when action=view, so one component of the cache ID will be the parameter ″action″ when the value equals ″view″. The news category is also an essential part of the cache ID. Finally, the optional session attribute for the user's layout is included in the cache ID. The cache-entry now looks like this:

   ```
   <cache-entry>
    <name> /newscontroller </name>
    <class>servlet  </class>
      <cache-id>
      <component id="action" type="parameter">
       <value>view</value>
       <required>true</required>
      </component>
      <component id="category" type="parameter">
       <required>true</required>
      </component>
      <component id="layout" type="session">
       <required>false</required>
      </component>
    </cache-id>
   </cache-entry>
   ```

3. Define dependency ID rules. For this servlet, a dependency ID will be added for the category. Later, when the category is invalidated due to an update event, all views of that news category will be invalidated. After adding our dependency-id, the cache-entry now looks like this:

   ```
   <cache-entry>
    <name>newscontroller </name>
    <class>servlet  </class>
      <cache-id>
      <component id="action" type="parameter">
       <value>view</value>
       <required>true</required>
      </component>
      <component id="category" type="parameter">
       <required>true</required>
   ```

```
    </component>
    <component id="layout" type="session">
     <required>false</required>
    </component>
   </cache-id>
  <dependency-id>category
   <component id="category" type="parameter">
    <required>true</required>
   </component>
  </dependency-id>
 </cache-entry>
```

4. Define invalidation rules. Since we defined a category dependency ID, we will now define an invalidation rule to invalidate the category when action=update. To incorporate the conditional logic, we will add "ignore-value" components into the invalidation rule. These components will not add to the output of the invalidation ID, but will only determine whether or not the invalidation ID is created and executed. The final cache-entry now looks like this:

```
<cache-entry>
 <name>newscontroller </name>
 <class>servlet  </class>
   <cache-id>
   <component id="action" type="parameter">
    <value>view</value>
    <required>true</required>
   </component>
   <component id="category" type="parameter">
    <required>true</required>
   </component>
   <component id="layout" type="session">
    <required>false</required>
   </component>
  </cache-id>
 <dependency-id>category
  <component id="category" type="parameter">
   <required>true</required>
  </component>
 </dependency-id>
 <invalidation>category
  <component id="action" type="parameter" ignore-value="true">
   <value>update</value>
   <required>true</required>
  </component>
  <component id="category" type="parameter">
   <required>true</required>
     </component>
 </invalidation>
</cache-entry>
```

Cache monitor

Cache monitor is an installable Web application that provides a real-time view of the current state of dynamic cache. You use it to help verify that dynamic cache is operating as expected. The only way to manipulate the data in the cache is by using the cache monitor. It provides a GUI interface to manually change data.

Cache monitor provides information on the cache in the Servant Region to which your browser connects to interact with the monitor. Thus, in an environment with multiple Servant Regions, Cache monitor provides a partial view of caching activity.

Cache monitor provides a way to:

- **Verify the configuration of dynamic cache**

The WebSphere Application Server adminstrative console provides ways to enable the dynamic cache service and configure properties, such as maximum size of the cache and disk offload location, as well as advanced features such as controlling external caches. Cache monitor offers a way for dynamic cache users to verify the configuration of the dynamic cache by providing a convenient view of the configured features and properties in the cache monitor.

- **Verify the cache policies**

  To cache an object, WebSphere Application Server must know how to generate unique IDs for different invocations of that object. This is performed by providing rules for each cacheable object in the `cachespec.xml` file, found inside the Web module `WEB-INF` or `enterprise bean META-INF` directory. Each cacheable object can have multiple cache ID rules that execute in sequence until either a rule returns a cache ID or no more rules remain to execute. If none of the cache ID generation rules produce a valid cache ID, then the object is not cached. Since there can be multiple `cachespec.xml` files with multiple cache ID rules, cache monitor provides a convenient way to verify the policies of each object. It offers a view of all the cache polices currently loaded in dynamic cache. This view is also convenient to verify that the `cachespec.xml` file was read by the dynamic cache without errors.

- **Monitor cache statistics**

  Cache monitor provides a view of the essential cache data, such as number of cache hits, cache misses, and number of entries in cache. This helps to tune the cache configuration optimally to get the best performance improvement out of dynamic cache. For example, if the number of used entries is often high, and entries are being removed and recreated, one might consider increasing the maximum size of the cache or enabling disk offload.

- **Monitor the data flowing through the cache**

  Once a cacheable object is invoked, dynamic cache creates a cache entry for it that contains the output of the execution and metadata, such as time to live, sharing policy, etc. Entries are distinguished by a unique ID string that is based on the rules specified in the `cachespec.xml` file for this objects name. Objects with the same name may generate multiple cache IDs for different invocations, based on request parameters and attributes for each invocation. Cache monitor provides a view of all the cache entries currently in cache, based on the unique ID. It also provides a view of the group of cache entries that share a common name (also known as template). Cache entries can also be grouped together by a dependency ID, which is used to invalidate the entire group of entries dependent on a common entity. Therefore, cache monitor also provides a view of the group of cache entries that share a common dependency ID.

  For each entry, cache monitor also displays metadata, such as time to live, priority and sharing-policy, and provides a view of the output that has been cached. This helps the customer to verify which pages have been cached, that the pages have been cached with the right attributes such as time to live, priority, etc., and that the pages have the right content.

- **Monitor the data in the edge cache**

  Dynamic cache provides support to recognize the presence of an Edge Side Include (ESI) processor and to generate ESI include tags and appropriate cache policies for edge cacheable fragments. The ESI processor has the ability to cache whole pages, as well as fragments, providing a higher cache hit ratio. There can be multiple ESI processors running on multiple hosts configured for caching.

  Cache monitor provides a list of all ESI processes and their hosts that are enabled for caching. It also provides a way to select a host or a processor, and view its edge cache statistics as well as current cache entries.

- **View the data offloaded to the disk**

By default, when the number of cache entries reaches the configured limit for a given server, eviction of cache entries occurs, allowing new entries to enter the cache service. The dynamic cache includes the disk offload feature that copies the evicted cache entries to disk for future access. Cache monitor offers a view of the content offloaded to disk that corresponds to the view of contents cached in memory.

- **Manage the data in the cache**

  Besides displaying cache content, cache monitor also provides some basic operations on the data in the cache:
  - Removing an entry from the cache
  - Removing all entries for a certain dependency ID
  - Removing all entries for a certain name (template)
  - Moving an entry to the front of the least recently used queue to avoid eviction
  - Moving an entry from the disk to the cache
  - Clearing the entire contents of the cache
  - Clearing the contents of the disk cache

  These functions are useful for dynamic cache customers, as they provide a way to manually change the state of the cache without having to restart the server.

Edge cache statistics

Cache monitor provides a view of the edge cache statistics.

The following statistics are available:

- **ESI Processors**. Number of processes configured as edge caches.
- **Number of Edge Cached Entries**. Number of entries currently cached on all edge servers and processes.
- **Cache Hits**. Number of requests that match entries on edge servers.
- **Cache Misses By URL.** A cache policy does not exist on the edge server, for the requested template.

  Note:
  - The initial ESI request for a template that has a cache policy on WebSphere Application Server will result in a miss.
  - Every request for a template that does not have a cache policy on WebSphere Application Server will result in a miss by URL on the edge server.
- **Cache Misses By Cache ID**. The policy for the requested template exists on the edge server, and a cache ID is created, based on the ID rules and the request attributes, but the cache entry for this ID does not exist.

  Note: If the policy exists on the edge server for the requested template, but a cache ID match is not found, based on the ID rules and the request attributes, it is not treated as a cache miss.
- **Cache Timeouts**. Number of entries removed from the edge cache, based on the timeout value.
- **Evictions**. Number of entries removed from the edge cache, due to invalidations received from WebSphere Application Server.

Troubleshooting the dynamic cache service

Complete the steps below to resolve problems that you think are related to the dynamic cache service.

1. Review the JVM logs for your application server. Messages prefaced with *DYNA* result from dynamic cache service operations.
   a. View the JVM logs for your application server. Each server has its own JVM log file. For example, if your server is named *Member_1*, the JVM log is located in the subdirectory *install_root*/logs/Member_1. To use the administration console to review the JVM logs, click **Troubleshooting > Logs and Trace >** ***server_name*** **> JVM Logs > Runtime > View**.
   b. Find any messages prefaced with *DYNA* in the JVM logs, and write down the message IDs. A sample message having the message ID *DYNA0030E* follows:

      ```
      DYNA0030E: "property" element is missing required attribute "name".
      ```
   c. Find the message for each message ID in the WebSphere Application Server InfoCenter. In the InfoCenter navigation tree, click ***product_name*** **> Reference > Messages > DYNA** to view dynamic cache service messages.
   d. Read the message **Explanation** and **User Action** statements. A search for the message ID `DYNA0030E` displays a page having the following message:

```
DYNA0030E: "property" element is missing required attribute "name".
  Explanation: A required attribute was missing in the cache configuration.
  User Action: Add the required attribute to your cache configuration file.
```

      This explanation and user action suggests that you can fix the problem by adding or correcting a required attribute in the cache configuration file.
   e. Try the solutions stated under **User Action** in the *DYNA* messages.
2. Use the cache monitor to determine whether the dynamic cache service is functioning as expected. The cache monitor is an installable Web application that displays simple cache statistics, cache entries, and cache policy information.
3. If you have completed the preceding steps and still cannot resolve the problem, contact your IBM software support representative. Use the collector tool (collector.bat or collector.sh located in the `bin` directory) to gather trace information and other configuration information for the support team to diagnose the problem. The collector tool gathers dynamic cache service files and packages them into a JAR file. The IBM representative can specify when and where to send the JAR file. The IBM representative might ask you to complete a diagnostic trace. To enable tracing in the administrative console, click **Troubleshooting > Logs and Trace >** *server_name* **> Diagnostic Trace** and specify **Enable trace with the following specification**. The IBM representative can tell you what trace specification to enter. Note that dynamic cache trace files can become large in a short period of time; you can limit the size of the trace file by starting the trace, immediately recreating the problem, and immediately stopping the trace.

   For current information available from IBM Support on known problems and their resolution, see the IBM Support page.

   IBM Support has documents that can save you time gathering information needed to resolve this problem. Before opening a PMR, see the IBM Support page.

Troubleshooting tips for the dynamic cache service

The dynamic cache service works within an application server Java virtual machine (JVM), intercepting calls to cacheable objects. This article describes some common run-time and configuration problems and remedies.

Servlets are not cached

Recommended response
: Enable servlet caching. On the Web container page of the administrative console, select the **Enable servlet caching** check box.

Cache entries are not written to disk

Explanation
: Cache entries are written to disk when the cache is full and new entries are added to the memory cache. Cache entries also are written to disk when the `flushToDiskOnStop` system property is set and the server is stopped.

Recommended response
: Verify that **Disk offload** is enabled on the Dynamic Cache Service page of the administrative console. Also verify that cache entries written to disk are serializable and do not have the `PersistToDisk` configuration set to `false`.

Some servlets are not replicated or written to disk

Recommended response
: Ensure that the attributes and response are serializable. If the you do not want to store the attributes, use the following property in your cache policy:

```
<property name="save-attributes">false</property>
```

Dynamic cache service does not cache fragments on the Edge

Recommended response
: Set the `EdgeCacheable` property to `true` in the cache policy for those entries that are to be cached on the Edge.

```
<property name="EdgeCacheable">true</property>
```

Dynamic cache invalidations are not sent to the IBM HTTP Server (IHS) plug-in

Explanation
: The `DynaCacheEsi.ear` file is required to send invalidations to external caches.

Recommended response
: Install `DynaCacheEsi.ear` using the administrative console.

Cache entries are evicted often

Problem
: The cache is full and new entries are added to the cache.

Explanation
: Cache entries are evicted when the cache is full and new entries are added to the cache. A LRU eviction mechanism removes the least recently used entry to make space for the new entries.

Recommended response
: Either enable **Disk offload** on the Dynamic Cache Service page of the administrative console so the entries are written to disk. Or, increase the cache size to accommodate more entries in the cache.

Cache entries in disk with timeout set to 0 expire after one day

Explanation
: The maximum lifetime of an entry in disk cache is 24 hours. A timeout of `0` in the cache policy configures these entries to stay in disk cache for one whole day, unless they are evicted earlier.

Recommended response Set the timeout for the cache policy to a number greater than `0`.

I cannot monitor cache entries on the Edge

Explanation Use the cache monitor for monitoring contents in memory cache, disk cache and external caches (Edge cache). For the ESI processor's cache to be visible in the cache monitor, the `DynaCacheEsi.ear` application must be installed and the `esiInvalidationMonitor` property must be set to `true` in the `plugin-cfg.xml` file.

Recommended response Install the DynaCacheEsi.ear application and set the `esiInvalidationMonitor` property to `true` in the `plugin-cfg.xml` file.

I want to cache static contents using the dynamic cache service

Explanation You can cache static contents using the dynamic cache service. Static contents in WebSphere application server are served by the `SimpleFileServlet` file.

Recommended response Create a cache policy for the class `com.ibm.ws.webcontainer.servlet.SimpleFileServlet.class` to cache static contents. It is advisable to use the dynamic cache service for caching more expensive dynamic contents than static contents.

I want to tune cache for my environment

Recommended response Use the Tivoli Performance viewer to study the caching behavior for your applications. Also, do the following:

- Increase the priority of cache entries that are expensive to regenerate.
- Modify timeout of entries so that they stay in memory as long as they are valid.
- Enable disk offload to store LRU evicted entries.
- Increase the cache size.

Chapter 16. Assembling applications with the AAT

Assemble application modules (known as EAR files) from new or existing J2EE 1.3 or 1.4 modules, including these archives: Web application archives (WAR), resource adapter archives (RAR), enterprise beans (EJB JAR), and application client archives (JAR). This packaging and configuration of code artifacts into application modules or stand-alone Web modules is necessary for deploying the applications onto the application server.

5.0.2 + For the Windows and Linux Intel operating systems, the Assembly Toolkit replaces the Application Assembly Tool (AAT). Visit the Web site `http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q =&uid=swg24005125&loc=en_US&cs=utf-8&lang=en+en` to download the Application Server Toolkit product, which offers the Assembly Toolkit and other products. The Assembly Toolkit consists of the J2EE Perspective of the WebSphere Studio Application Developer product without the code generation capabilities.

Gather the code artifacts that you want to package into one or more assembled modules. Code artifacts include these items that you have created and unit tested in your favorite integrated development environment:

- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Other supporting classes and files

1. Start the Application Assembly Tool (AAT)on Windows.
2. Migrate existing J2EE 1.2 modules to J2EE 1.3. The AAT has an option for migrating J2EE 1.2 application modules to J2EE 1.3. The J2EE 1.2 module is kept intact, with a new 1.3 module created. See also the earconvert tool documentation.

   You must migrate J2EE 1.2 application modules to which you want to add J2EE 1.3 level WAR, RAR, EJB and client modules. This tool migrates only the application modules. J2EE 1.2-level modules inside a J2EE 1.2 application module must be migrated by other means.
3. Assemble new EJB modules (enterprise bean JAR files) as needed. Assemble an EJB module to contain enterprise beans and related code artifacts. (Group Web components, client code, and resource adapter code in separate modules.)

   You can install an EJB module as a stand-alone application or you can combine it with other modules into an enterprise application.
4. Assemble new Web modules (WAR files) as needed. Assemble a Web module to contain servlets, JSP files, and related code artifacts. (Group enterprise beans, client code, and resource adapter code in separate modules.)

   You can install a Web module as a stand-alone application or combine it with other modules into an enterprise application.
5. Assemble new application client modules (client JAR files) as needed.
6. Assemble new resource adapter archives (RAR files) as needed. Assemble a resource adapter archive module to contain the library implementation code that your application uses to connect to enterprise information systems (EIS). (Group enterprise beans, Web components, and client code in separate modules.)

© Copyright IBM Corp. 2003

7. Assemble an application module from other module types. You are ready to combine your new or migrated modules into an application module (EAR file). For applications containing only Web modules, this step is optional. It is feasible to deploy Web modules without assembling them into application modules.
8. Verify your archive files. Verify your archive files and correct any problems so that generation of deployment code is successful. During verification, the AAT checks that an archive file is complete, and that deployment descriptor properties and references contain appropriate values.
9. Remember to save your application one last time.
10. Generate code for deployment for applications containing EJB modules. If the application modules contain EJB modules, you must generate deployment code for the enterprise beans in the application before you deploy applications on the server. The AAT provides this ability, or you can use the ejbdeploy command line tool.
11. Open existing modules (**File > Open**) in the AAT to modify them as needed. For example, you can add or remove modules and edit deployment descriptor properties.

After assembling your applications, use a systems management tool to deploy the EAR or WAR files onto the application server.

The systems management tool follows the security and deployment instructions defined in the deployment descriptor, and enables you to modify bindings specified within the AAT. The tool locates the required external resources that the application uses, such as enterprise beans and databases.

Select a tool to use:
- Deploying and managing applications with the GUI
- Deploying and managing applications using programming
- Deploying and managing applications using scripting

If you are uncertain of which systems management tool to use, try using the administrative console.

Application assembly and J2EE applications

Application assembly is the process of creating an Enterprise Archive (EAR) file containing all files related to an application, as well as an XML deployment descriptor for the application. This configuration and packaging prepares the application for deployment onto an application server.

EAR files are comprised of the following archives:
- Enterprise bean (JAR) files (known as EJB modules)
- Web application (WAR) files (known as Web modules)
- Application client (JAR) files (known as client modules)
- Resource adapter (RAR) files (known as resource adapter modules)

Ensure that modules are contained in an EAR file so that they may be deployed onto the server. The exceptions are WAR modules, which you can deploy individually. Although WAR modules can contain regular JAR files, they cannot contain the other module types described previously.

The assembly process includes the following:
- Selecting all of the files to include in the module

- Creating a deployment descriptor containing instructions for module deployment on the application server.

  As you configure properties using the Application Assembly Tool (AAT), the tool generates the deployment descriptor for you. While the AAT graphical interface is recommended, you can also edit descriptors directly in your favorite XML editor.
- Packaging modules into a single Enterprise archive (EAR) file, which contains one or more files in a compressed format

Archive support in Version 5.0

These archives and Web components are supported:
- J2EE 1.3 Enterprise application (EAR) files
- EJB 2.0 (JAR) files
- Servlet 2.3 Web application WAR files
- Application Client 1.3 JAR files
- Connector 1.0 RAR files

These archive files and Web components are back-level and may be read but not created or changed:
- J2EE 1.2 EAR files
- EJB 1.1 JAR files
- Servlet 2.2 WAR files
- Application Client 1.2 JAR files

Starting the Application Assembly Tool (AAT)

A graphical interface is available for packaging code artifacts into various archives (modules) and configuring their J2EE 1.3 compliant deployment descriptors. The Application Assembly Tool (AAT) is available from the Windows Start menu, or you can invoke the tool from a command line as described in the Steps for this task.

On z/OS, the user must FTP (in binary) the setup.exe file from the AppServer/lib directory and install it on their Windows platform. This support is for the Windows platform Only.

If you access the Application Assembly Tool from a remote browser and select the Help, the Help files do not display. You can only view the Help files from a locally installed browser. To view the Help files and avoid this problem, close all the Netscape sessions on the remote machine and click **Help**. A new Netscape session starts, and you can then view the Help files.

1. Change directory at a system command prompt to the location of the `assembly.bat|sh` file, typically `install_root/bin`.
2. Run the assembly script to launch the graphical interface.
3. Select whether to work with an existing module or create a new one.

The navigation tree displays a hierarchical structure used to build the contents of a new module, or to work with the contents of an existing module. Icons in the tree represent the components, assembly properties, and files for the module. The assembly properties appear in the AAT workspace.

Consider whether you have any existing J2EE 1.2 application modules that you would like to migrate to J2EE 1.3.

You can create new modules of the following types, to assemble into an application module later:
- Assembling EJB modules
- Assembling Web modules
- Assembling application client modules
- Assembling resource adapter modules

Rather than create new modules to assemble an application, you can proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

Migrating application modules from J2EE 1.2 to J2EE 1.3

The Application Assembly Tool (AAT) has an option for migrating J2EE 1.2 application modules to J2EE 1.3. The J2EE 1.2 module is kept intact, with a new 1.3 module created. See also the earconvert tool documentation.

Migrate J2EE 1.2 application modules to which you want to add J2EE 1.3 level Web application (WAR) modules, Resource adapter (RAR) modules, Entity bean (EJB) modules, and application client modules. This tool migrates only the application modules. Migrate J2EE 1.2-level modules inside a J2EE 1.2 application module by other means.

Note: When Entity beans are moved from a J2EE 1.2 module to a J2EE 1.3 module, the EJB container will then apply rules defined in the EJB 2.0 specification to these beans. The EJB 2.0 specification mandates that when a `findBy` method is called on a bean home (except for `findByPrimaryKey`), the EJB container must cause other Entity beans enlisted in the same transaction to write out their current state to the persistent store. This is to ensure that the `findBy` operation is performed on the most current data. Application developers should plan for and be aware of any changes to the application behavior as a result of this rule.

1. Start the AAT.
2. Use it to open the J2EE 1.2 application module you want to migrate.
3. Click **Convert EAR** from the file menu.
4. Save the new J2EE 1.3 application.

Assemble zero or more new modules of your choice:
- Assembling EJB modules
- Assembling Web modules
- Assembling application client modules
- Assembling resource adapter modules

Another option is to proceed directly to assembling a new application module. You can create any new modules that you need, while assembling an application module.

earconvert tool

A command line tool is provided for migrating J2EE 1.2 application modules to J2EE 1.3. This migration enables you to add J2EE 1.3 modules to the migrated application module.See also the Application Assembly Tool (AAT) for information on performing this task.

Migrate J2EE 1.2 application modules to which you want to add J2EE 1.3 level Web application (WAR), Resource adapter (RAR), Enterprise beans (EJB), and client modules. This tool migrates only the application modules. Ensure that you migrate J2EE 1.2-level modules inside a J2EE 1.2 application to prevent working with back-level files.
The IBM WebSphere Client Development Kit for z/OS is located in the `setup.exe` file of the *install_root*/lib directory. To install the IBM WebSphere Client Development Kit for z/OS, use the File Transfer Protocol (in binary) to transfer the `setup.exe` file to the machine on which the Windows operating system is installed, and run the `setup.exe` file.

earconvert
j2ee_1.2_file_name

j2ee_1.3_file_name

Parameters

Supported arguments include:

"j2ee_1.2_file_name"
Specifies the actual name of the existing J2EE 1.2 application file. (In this and other arguments, use quotation marks to allow for path names that contain spaces.)

"j2ee_1.3_file_name"
Specifies what you would like to name the new J2EE 1.3 application file.

The following command creates a new J2EE 1.3 archive, `new_application.ear`, based on the J2EE 1.2 archive, `existing_application.ear`.

earconvert `existing_application.ear new_application.ear`

Assembling new or modifying existing modules

Ensure that code artifacts, such as servlets, JSP files, enterprise beans, and application clients are assembled into their respective modules.

If you want to use existing J2EE 1.2 modules in your J2EE 1.3 application, migrate these modules to J2EE 1.3 first. Also migrate any J2EE 1.2 application modules to which you want to add J2EE 1.3 modules.

You are now ready to combine your new or migrated modules into an application module Enterprise application (EAR file).

The Application Assembly Tool (AAT) provides flexibility in assembling applications from various Web application (WAR), Resource adapter (RAR), Enterprise beans (EJB JAR), and application client (JAR) files. Options described in assembling applications include:
- Importing an existing module (JAR, RAR or WAR file)
- Creating a new module while you create the new application
- Copying code artifacts, such as servlets, from one module to another of the same type, to reside in the new application

1. Start the AAT.
2. From the New tab, select **Application**, and click **OK**, if you did not already specify to create a new application module. Each of the next three steps is optional, but you must perform at least one of them.
3. Import existing modules into the application module.

a. Right-click the folder for the type of module you want to import, such as an EJB module, in the navigation tree.
b. Click **Import** from its right-click menu.
c. Use the file browser to locate and select the archive file for the module.
d. Click **Open**. The archive file appears under the appropriate folder in the navigation tree.
e. Click the plus sign **(+)** next to the icon for the archive, to view the module contents and edit its properties if needed.
f. Save the application module.

4. Create a new archive file to include in the application.
   a. Right-click the folder for the type of module to create (such as enterprise beans (EJB) modules, Web application modules (WAR), resource adapter (RAR) files, or application client modules) in the navigation tree.
   b. Click **New** from its right-click menu.
   c. Configure properties of the new module when it displays.
   d. Click **OK**. The archive file displays under the appropriate folder.
   e. Click the plus sign **(+)** to verify file contents and enter assembly properties.
   f. Add enterprise beans, if this is an EJB module.
   g. Right-click the folder corresponding to the type of bean to create (session bean or entity bean), and click **New** or **Import**.
   h. Configure properties of the enterprise bean when it displays.
   i. Click **OK**. The enterprise bean appears in the navigation pane.
   j. Click the plus sign **(+)** to verify file contents and enter assembly properties.
   k. Save the application module.
5. Copy code artifacts, such as servlets, from one module to another of the same type, to reside in the new application.
   a. Identify the code artifact to copy, and the type of module in which it resides. Make sure you already have the same kind of module (such as a Web module) created in the new application module.
   b. Open a separate, existing module in the AAT by selecting **File** > **Open** from the menu bar.
   c. Arrange the AAT workspace so that you can see both the new application module and the source archive containing the code artifact.
   d. Copy and paste the code artifact from the source module to the same module type in the new application. For example, copy a container-managed persistence (CMP) bean from the source EJB module into the new EJB application module.
   e. Save the application module.
6. Continue to add desired modules to the application module.
7. Define security properties for the application.
   a. Right-click the **Security Roles** icon in the navigation tree.
   b. Click **New**.
   c. Configure the security properties.
   d. Click **OK**.
8. Add supplementary files needed by the application.
   a. Right-click the **Files** icon in the navigation tree, and select **Add Files**.
   b. Add files, using the Add Files dialog.
9. Save the application module.

You are performing application assembly results in a J2EE 1.3 compliant EAR file containing one or more WAR, RAR, or JAR files.

Note: If you use the Application Assembly Tool to create application client modules, you must also use the Application Client Resource Configuration Tool. Using this tool, you can define references to resources (other than enterprise beans) on the machine where the application client resides.

After assembling an application, you can do the following:
1. Verify archive files.
2. Generate code for deployment.
3. Use the administrative console to install the application onto an application server.

   Note: If your application has a large number of modules, it might not install successfully onto a server. Package your application so the `.ear` file has as few modules as are necessary. Modules can include metadata for the modules such as information on deployment descriptors, bindings and IBM extensions.
4. Use the administrative console at installation time to carry out the security instructions defined in the deployment descriptor and to locate required external resources, such as enterprise beans and databases. You can add configuration properties and redefine binding properties defined in the Application Assembly Tool.
5. After the application deploys, use the Application Assembly Tool to modify the application by adding or removing modules, editing deployment descriptor properties and regenerating code for deployment.

Adding files to assembled modules

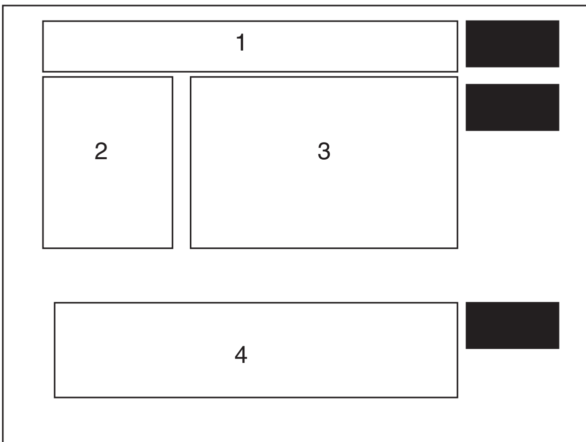
Review the usage scenario (as follows) to become familiar with the **Add Files** dialog.

Use the **Add Files** dialog box of the Application Assembly Tool (AAT) to import files into assembled modules including Enterprise application (EAR), Web application (WAR), Resource adapter (RAR) and Application client (JAR) files.

This task assumes that you are performing another task, such as assembling a Web or EJB module, when the **Add Files** dialog is presented to you.
1. Click **Browse**. Locate the files to add.
   - To add specific individual files, select the directory or archive (WAR, JAR, RAR, ZIP, for example) containing the files.
   - To add an entire directory of files, select its parent directory.
2. Click **OK**. The selected directory or achive appears in the top left part of the dialog box, in an expandable tree.

   The top right part of the dialog box shows the contents of the directory, subdirectory, or archive that is selected on the left-hand side.
3. Select one or more items to add from the top right part of the dialog, then click **Add**. As you add files, they will be displayed in the lower half of the dialog box.
4. Change your left-hand selection to gain access to other files that you want to add, as needed.
5. Click **OK** when all of the files that you want to add appear in the lower half of the dialog box.

The following example refers to the main areas of the Add Files dialog box. Details such as clicking OK are omitted. Refer to the detailed task steps above for this information.



Suppose you are constructing a new application module and want to add `myFile.txt` file to the archive as a supplementary file. The `myFile.txt` currently is contained within the myFiles subdirectory of a JAR file that resides somewhere on your directory system.

1. Browse for the JAR file.
2. Select the JAR file.
3. Exit the browse dialog. At this point:
   - Area 1 of the **Add Files** dialog contains the path to the JAR file.
   - Area 2 displays the JAR file name as the root directory of an expandable tree showing the directories in the JAR file -- including `myFiles`.
   - Area 3 shows the root contents of the JAR file, as well as any subdirectories visible from the root.
4. Select the `myFiles` directory from area 2, causing `myFile.txt` to become visible in area 3.
5. Click **myFiles.txt** from area 3 and specify to **Add** the file. Now this file is listed in area 4, the lower half of the dialog, which indicates it is the file that you want to add to the new application module.
6. Exit the **Add Files** dialog.

Resource environment reference assembly settings

Resource environment reference elements contain declarations of an enterprise bean's reference to an administered object associated with a resource in the enterprise bean's environment.

Name

Specifies the name of the resource environment reference.
Its value is the environment entry name used in the enterprise bean code.

Data type	String

Description

Contains the information that the EJB jar file producer wants to provide to the EJB jar file consumer.

Data type	String

Type

Specifies the type of a resource environment reference.

Data type	String

Resource Adapter Archive file assembly settings

Use this page to set the resource adapter archive file properties.

File name

Specifies the file name of the Resource Adapter Archive.

Data type	String

Display name

Specifies a short name that is intended to be displayed by the GUI.

Data type	String

Description

Specifies a description that should include any information that the component file producer wants to provide to the consumer of the component file (that is, to the deployer).

Data type	String

EIS type

This helps in identifying EIS instances that can be used with this resource adapter.

Data type	String

Vendor name

Specifies a string-based version of the resource adapter from the resource adapter provider.

Data type	String

Version

Specifies a string-based version of the resource adapter from the resource adapter provider.

Data type	String

Specification

Specifies the version of the connector architecture specification that is supported by this resource adapter.

Data type	String

License required

Specifies if a license is or is not required.

Description

If a license is required, this field specifies the licensing requirements for the resource adapter module. For example, duration of license, number of connection restrictions, and so forth.

Data type	String

Implementation

The element (managedconnectionfactory class) that specifies the fully qualified name of the Java class that implements the javax.resource.spi.Managed-ConnectionFactory interface.

Data type	Class

Interface

The element (credential-interface) that specifies the interface that the resource adapter implementation supports for the representation of the credentials.
The possible values are:

<credential-interface> javax.resource.spi.security.PasswordCredential
</credential-interface>
<credential-interface> javax.resource.spi.security.GenericCredential
</credential-interface>

Data type	Class

Implementation

The element (connectionfactory class) that specifies the fully-qualified name of the ConnectionFactory class that implements the resource adapter specific ConnectionFactory interface.

Data type	Class

Interface

The element (connection-interface) that specifies the fully-qualified name of the Connection interface supported by the resource adapter.

Data type	Class

Implementation

The element (connection class) that specifies the fully-qualified name of the Connection class that implements the resource adapter specific Connection interface.

Data type	Class

Support Reauthentication

Specifies whether the resource adpater implementation supports re-authentication of existing ManagedConnection instances.
The values are either True or False.

Data type	String

Transaction

Specifies the level of transaction support provided by the resource adapter.
The three possible values are:
- NoTransaction
- LocalTransaction
- XATransaction

Data type	String

Small Icon

The image is used as an icon to represent the module in a GUI.
Specifies a JPEG or GIF file containing a small image (16x16 pixels).

Data type	Image

Large Icon

The image is used as an icon to represent the module in a GUI.
Specifies a JPEG or GIF file containing a small image (32x32 pixels).

Data type	Image

Basic Password

The basic user password authentication mechanism that is specific to an EIS.

Credential Interface
: Specifies the interface that the resource adapter implementation supports for the representation of the credentials. For Basic Password the credential value is *javax.resource.spi.security.PasswordCredential*.

Description
: Any information that describes Basic Password selection.

Data type	String

Kerboros V5

Specifies a Kerboros version 5 authentication mechanism.

Credential Interface
: Specifies the interface that the resource adapter implementation supports for the representation of the credentials. For Kerboros version 5, the credential value is *javax.resource.spi.security.GenericCredential*.

Description
: Any information that describes the Kerboros V5 selection.

Data type	String

Property Name

Specifies the name of a configuration property.
The possible values are:

<config-property-name>ServerName</config-property-name>
<config-property-name>PortNumber</config-property-name>
<config-property-name>UserName</config-property-name>
<config-property-name>Password</config-property-name>
<config-property-name>ConnectionURL</config-property-name>

Data type	String

Property Type

Contains the fully-qualified Java type of a configuration property as required by the ManagedConnectionFactory instance.

Data type	String

Property Value

Contains the value of a configuration entry.

Data type	String

Description

Describes the parent element.

Data type	String

Permission Specification

Specifies a security permission that is required by the resource adapter code.

Data type	String

Saving applications after assembly

Periodically save modules that you assemble with the Application Assembly Tool (AAT). Save any changes right before you close the module with which you are working.

This task assumes you have started the AAT and are working with a particular module.

1. Save the archive file by clicking **File** > **Save As**.
   - If you are saving an existing archive file or application, click **File** > **Save**.
2. Name the new archive file or application whatever you like. This step is optional if you are working with an existing archive file or application.

Now that you have saved your assembled application, you can verify your archives and generate code for deployment.

Verifying archive files

Verify your archive files and correct any problems so that generation of deployment code is successful. During verification, the Application Assembly Tool (AAT) checks that an archive file is complete, and that deployment descriptor properties and references contain appropriate values.

This task assumes you have previously assembled and saved one or more modules.

1. Start the Application Assembly Tool (AAT).
2. Click **File** > **Open** and select the module to verify.
3. Right-click the name of the module at the top of the navigation pane and click **Verify**.
4. Click **Verify** in the **Verify** window. The tool displays a scrolling window for viewing status messages as the verification proceeds.

5. Save the application.

Archive files have been verified. The following list includes, but is not limited to, areas that the verification process has checked:
- Required deployment properties contain values.
- Values specified for environment entries match their associated Java types.
- In both Enterprise application (EAR) and Web application (WAR) files:
  – The target enterprise bean of the link exists for EJB references.
  – The target role exists for security role references.
  – Security roles are unique.
- Each module listed in the deployment descriptor exists in the archive for EAR files.
- Files for icons, servlets, error and welcome pages listed in the deployment descriptor have corresponding files in the archive for WAR files.
- For EJB modules:
  – All class files referenced in the deployment descriptor exist in the JAR file.
  – Method signatures for enterprise bean home, remote and implementation classes are compliant with the EJB 2.0 specification.

If your application module contains EJB modules, generate code for deployment.

Otherwise, you are ready to deploy this application module (or stand-alone Web module) onto the application server.

Application assembly performance checklist

Application assembly tools are used to build J2EE components and modules into J2EE applications. Generally, assembling consists of defining application components and their attributes including enterprise beans, servlets and resource references. Many of these application configuration settings and attributes play an important role in the run-time performance of the deployed application. Use the following information as a check list of important parameters and advice for finding optimal settings:
- EJB modules
  – Entity bean Bean Cache - Activate at and Bean Cache - Load at settings
  – Method extensions Isolation level and Access intent settings
  – Container transactions assembly settings
- Web modules
  – Web modules assembly settings
    - Distributable
    - Reload interval
    - Reload enabled
- Web components
  – Load on startup

Generating code for deployment

Before deploying applications on the server, if the application modules contain EJB modules, you must generate deployment code for the enterprise beans in the application. The Application Assembly Tool (AAT) provides this ability, or you can use the ejbdeploy command line tool.

This task assumes you have already assembled an EJB module, added it to an application module, saved the application module, and verified the application module.

Before installing your application in WebSphere Application Server, you must generate deployment code for the application. This step is required for EJB modules and for any Enterprise application (EAR) files that contain EJB modules. During code generation, the Application Assembly Tool invokes the EJBDeploy tool to prepare entity bean (JAR) files for deployment in run time environment. To deploy a J2EE application, you can install the application in the administrative console.

The following steps assume that you are using the Application Assembly Tool to generate code for deployment.

1. Start the Application Assembly Tool (AAT).
2. Open the EAR or JAR file for which you want to generate code for deployment.
3. Click **File** > **Generate code for deployment** from the menu bar.
4. Specify the options for the server to use for generating code for the application deployment.

   Note: For Container managed persistence (CMP) entity beans, if the JAR file that you opened (`inputJar` file) contains a map and schema document, that schema is used. If the JAR file does not contain a map and schema document, the Application Assembly Tool uses a top-down mapping to generate files that contain mapping and database schema information.
5. Click **Generate Now**. Review the messaging box for details of any error that might occur.

   Note: Do not change the default output file name to be the same as the input filename, as the AAT cannot read and write to the same file name, and therefore, an error will occur.

After deployment code is generated for an application, the deployable archive is renamed with the prefix `Deployed_`.

Install the application on your server machine.

Note: Before deploying the application in your run time environment, you might need to set classpaths.

ejbdeploy tool

You can generate code for deployment by either using the Application Assembly Tool (AAT) or by using the Deployment Tool for Enterprise Java Beans (ejbdeploy) from a command prompt. For example, the options that you are able to set in AAT correspond with commands that the EJBDeploy tool uses to generate code for deploying an application.
For a detailed list of available options in the EJBDeploy tool, enter `ejbdeploy` from a command prompt.

For the z/OS environment: Use the `-e` option to display console output in ASCII characters rather than EBCDIC characters. Thus, enter `./ejbdeploy.sh -e` to see a list of available options and use the syntax `./ejbdeploy.sh -e <input> <working> <output>` and so on.

ejbdeploy syntax -- relationship to Application Assembly Tool options

Application Assembly Tool options	EJBDeploy tool options
Deployed module location	`outputJar`
Working Directory	`workingDirectory`
Dependent classpath	`cp`
Code generation only	`codegen`
Verify archive (unchecked)	`novalidate`
RMIC options	`rmic options`
Database type	`dbvendor`
Database name	`dbname`
Schema name	`dbschema`

Application Assembly Tool: Resources for learning

Use the following links to find relevant supplemental information about the Application Assembly Tool. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- Programming specifications
- Administration

Programming specifications
- J2EE 1.3 specification
- EJB specifications
- Servlet specifications

Administration
- Application Client files
- Connector RAR files

Chapter 17. Assembling applications with the Assembly Toolkit

Assemble enterprise application modules (EAR files) from new or existing Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 or 1.3 modules, including these archives: Web application archives (WAR), resource adapter archives (RAR), enterprise bean (EJB) JAR files, and application client archives (JAR). This packaging and configuration of code artifacts into application modules or stand-alone Web modules is necessary for deploying the applications onto the application server.

The Assembly Toolkit replaces the Application Assembly Tool (AAT). The Assembly Toolkit consists of the J2EE Perspective of the WebSphere Studio Application Developer product. With the Assembly Toolkit, you can create and modify J2EE applications and modules, edit deployment descriptors, and map databases.

The Assembly Toolkit is one of the tools provided by the Application Server Toolkit (ASTK). Follow instructions available with the ASTK to install the Assembly Toolkit.

5.0.2 Visit the Web site `http://www.ibm.com/support/docview.wss?rs=180&context=SSEQTP&q=ASTK&uid=swg24005125&loc=en_US` to download the Application Server Toolkit (ASTK) product, which offers the Assembly Toolkit and other products.

Gather the code artifacts that you want to package into one or more assembled modules. Code artifacts include these items that you have created and unit tested in your favorite integrated development environment:
- Enterprise beans
- Servlets, JavaServer Pages (JSP) files and other Web components
- Resource adapter (*connector*) implementations
- Application clients
- Other supporting classes and files

The Assembly Toolkit provides extensive online documentation. The articles on Assembly Toolkit provided in this InfoCenter supplement that documentation.

1. Start the Assembly Toolkit.
2. Optional: Read the online documentation for the Assembly Toolkit.
   - Read the section **Assembly Tool** on the Welcome to the Application Server Toolkit page. To access this page, click **Help > Welcome > Application Server Toolkit**.
   - Click **Help > Help Contents > Assembly Toolkit information**. The displayed documentation provides extensive information about the Assembly Toolkit.
   - Press F1 to access information specific to an Assembly Toolkit view or window.
   - Visit the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click

WebSphere Studio Application Developer > J2EE development. The documentation in the WebSphere Studio InfoCenter is similar to that in the Assembly Toolkit online information.

- See the article ″“Assembly Toolkit: Resources for learning” on page 896″ for additional sources.

3. Optional: Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
4. Optional: Open the J2EE Hierarchy view. Click **Window > Show View > J2EE Hierarchy**. Other helpful views include the Project Navigator view (**Window > Show View > Other > J2EE > Project Navigator**) and the Navigator view (**Window > Show View > Navigator**).
5. Migrate EAR, WAR, enterprise bean JAR files, application client JAR files, or resource adapter RAR files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import the files to the Assembly Toolkit.
6. Optional: Migrate a project from J2EE 1.2 to J2EE 1.3 using the J2EE Migration wizard. As part of the migration, you can migrate CMP 1.*x* beans to CMP 2.*x* beans. The J2EE Migration wizard is similar to the `earconvert` batch utility or the **File > ConvertEar** option of the AAT.
   a. In the J2EE Hierarchy view, right-click the enterprise application project (EAR file) you want to migrate.
   b. Click **Migrate > J2EE Migration Wizard**.
   c. Follow the instructions in the wizard.
7. Create an enterprise application project to which you can add archive files. You can create an enterprise application project separately or when you create archive files such as the following:
   - Create a Web project.
   - Create an application client.
   - Create an enterprise bean (EJB) project.
   - Create a resource adapter (connector) project.
8. Edit the deployment descriptors as needed. You can edit deployment descriptors for enterprise application, Web, application client, and enterprise bean (EJB) modules.
9. Optional: Generate enterprise bean (EJB) to relational database (RDB) mappings for EJB modules.
10. Verify the archive files.
11. Generate code for deployment for EJB modules or for enterprise applications that use EJB modules.
12. Generate code for deployment for Web services-enabled modules or for enterprise applications that use Web service modules.
13. Optional: Test your completed module on a WebSphere Application Server installation. Right-click a module, click **Run on Server**, and follow the instructions in the displayed wizard. Note that **Run on Server** works on the Windows, Linux/Intel, and AIX operating systems only; you cannot deploy remotely from the Assembly Toolkit to a WebSphere Application Server installation on a UNIX operating system such as Solaris.

    Important

    Important: Use **Run On Server** for unit testing only. Application Server Toolkit controls the WebSphere Application Server installation and, when an application is published remotely, the Toolkit overwrites

the server configuration file for that server. Do not use on production servers.

For instructions on remote testing, see the article "Setting Up a Remote WebSphere Application Server in WebSphere Studio V5" at http://www7b.boulder.ibm.com/wsdd/techjournal/0303_yuen/yuen.html.

After assembling your applications, use a systems management tool to deploy the EAR or WAR files onto the application server. The systems management tool follows the security and deployment instructions defined in the deployment descriptor, and enables you to modify bindings specified within the Assembly Toolkit. The tool locates the required external resources that the application uses, such as enterprise beans and databases.

Select a tool to use:
- Administrative console installation pages (GUI)
- Java administrative programs (programming)
- wsadmin AdminApp `install` command (scripting)

If you are uncertain of which systems management tool to use, try using the administrative console.

If your application has a large number of modules, it might not install successfully onto a server. Package your application so that the `.ear` file contains necessary modules only. Modules can include metadata for the modules such as information on deployment descriptors, bindings, and IBM extensions.

Use the administrative console at installation to complete the security instructions defined in the deployment descriptor and to locate required external resources, such as enterprise beans and databases. You can add configuration properties and redefine binding properties defined in the Assembly Toolkit.

Application assembly and J2EE applications

Application assembly is the process of creating an enterprise archive (EAR) file containing all files related to an application, as well as an XML deployment descriptor for the application. This configuration and packaging prepares the application for deployment onto an application server.

EAR files are comprised of the following archives:
- Enterprise bean JAR files (known as EJB modules)
- Web archive (WAR) files (known as Web modules)
- Application client JAR files (known as client modules)
- Resource adapter archive (RAR) files (known as resource adapter modules)

Ensure that modules are contained in an EAR file so that they can be deployed onto the server. The exceptions are WAR modules, which you can deploy individually. Although WAR modules can contain regular JAR files, they cannot contain the other module types described previously.

The assembly process includes the following actions:
- Selecting all of the files to include in the module.
- Creating a deployment descriptor containing instructions for module deployment on the application server.

  As you configure properties using the Assembly Toolkit, the tool generates the deployment descriptor for you. While the Assembly Toolkit graphical interface is recommended, you can also edit descriptors directly in your favorite XML editor.

- Packaging modules into a single EAR file, which contains one or more files in a compressed format.

Archive support in Version 5.0

The following archives and Web components are supported in Version 5.0:
- Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 and 1.3 enterprise archive (EAR) files
- Enterprise bean (EJB) 2.0 JAR files
- Servlet 2.3 Web archive (WAR) files
- Application client 1.2 and 1.3 JAR files
- Connector 1.0 resource adapter archive (RAR) files

These archive files and Web components are back-level and can be read but not created or changed:
- J2EE 1.2 EAR files
- EJB 1.1 JAR files
- Servlet 2.2 WAR files
- Application client 1.2 JAR files

Starting the Assembly Toolkit

The Assembly Toolkit provides a graphical interface for packaging code artifacts into various archives (modules) and configuring related Java 2 Platform, Enterprise Edition (J2EE) Version 1.2 or 1.3 compliant deployment descriptors.

The Assembly Toolkit is a component of the Application Server Toolkit (ASTK). To install the Assembly Toolkit, follow the installation instructions for the ASTK and, when prompted by the ASTK installation program, select to install the application assembly toolkit.

Users on the z/OS platform: The Assembly Toolkit is not available on the z/OS platform. FTP in binary the `setup.exe` file from the `AppServer/lib` directory and install the `setup.exe` file on an operating system that the ASTK supports, such as a Windows platform.

If you have installed the Assembly Toolkit component of ASTK previously and you install the Assembly Toolkit again, you must delete the workspace of the previous installation of ASTK before starting the Assembly Toolkit. The default workspace directory is `*my_directory*\IBM\astk\workspace`. If you do not delete the workspace for a previous installation of ASTK, you might encounter error messages such as the following when starting the Assembly Toolkit:

```
Problems during startup. Check the ".log" file in the ".metadata"
directory of your workspace.
```

1. Run the `astk` executable.
2. In the Application Server Toolkit window, specify the workspace directory and click **OK** to launch the graphical interface.

The navigation tree displays a hierarchical structure used to build the contents of a new module, or to work with the contents of an existing module.

Consider whether you have any existing J2EE 1.2 application modules that you would like to migrate to J2EE 1.3.

You can import or create new modules of the following types, to assemble into an application module later:
- Assembling enterprise bean (EJB) modules
- Assembling Web modules
- Assembling application client modules
- Assembling resource adapter modules

Rather than import or create new modules to assemble an application, you can proceed directly to assembling a new application module. While assembling an application module, you can create any new modules that you need.

astk command

The **astk** command starts the Application Server Toolkit. The command file is the astk executable file.

Location of the command file

The astk executable file resides in the main installation directory for Application Server Toolkit (ASTK).

Command syntax

- Issue the **astk** command:

  ```
  ASTK_install_root/astk
  ```

  Or, double-click the **ASTK** icon.
- The **astk** command has no options or command-line parameters.

Usage notes

- Is the astk executable file read-only?

  Yes
- Is this file updated by a product component?

  No
- How and when are the contents of this file used?

  The `astk` executable file provides the Assembly Toolkit component for the Application Server Toolkit. Run the `astk` executable file to start the Assembly Toolkit. There are no parameters or command-line options.

Migrating code artifacts to the Assembly Toolkit

You can migrate enterprise archive (EAR), Web archive (WAR), enterprise bean JAR, application client JAR, resource adapter archive (RAR) files created with the Application Assembly Tool (AAT) or a different tool to the Assembly Toolkit. To migrate files, import the files to the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Import an enterprise application.
3. Import a WAR file.
4. Import an application client file.
5. Import an enterprise bean JAR file.
6. Import a resource adapter RAR file. RAR files are also known as *connectors*.
7. Verify the archive files.
8. Generate code for deployment.

Importing enterprise applications

You can import an enterprise archive (EAR) file and define a new enterprise application project using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Enterprise Applications** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the EAR file and drop it on a view.
3. In the Import dialog, specify the EAR file to import and the project name:
   a. Click **EAR file > Next**.
   b. Specify the EAR file to import. Use **Browse** to locate the EAR file and specify its full path name.
   c. Optional: Specify a new enterprise application project name. A project name is assigned automatically. The project name you specify must be unique within the directory.
   d. Click **Finish**.
4. Verify the contents of the new enterprise application project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and view the new project.
   - Click **Window > Show View > Navigator** to see the associated files for the enterprise application project in a Navigator view.

Importing WAR files

Your can import a Web application archive (WAR) file and define a new enterprise archive (EAR) project and Web module for the WAR file using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Web Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the WAR file and drop it on a view.
3. In the Import dialog, specify a WAR file and a Web project name:
   a. Click **WAR file > Next**.
   b. Specify a WAR file. Use **Browse** to locate the WAR file and specify its full path name.
   c. Specify a Web project. For example, if you are importing the `HelloWorld.war` file, you might name the project `HelloWorld`. Click **New** and specify `HelloWorld` for the project name.
   d. Optional: To add the WAR file and Web project to an enterprise application, enable **Configure advanced options** and click **Next**. On the J2EE Settings page, specify the EAR project, the context root (Web project), and the J2EE 1.2 or 1.3 specification to use for the Web module. The J2EE 1.3 specification (the default) includes the Servlet 2.3 specification and the JSP 1.2 specification; applications developed for the J2EE 1.3 specification typically target a WebSphere Application Server Version 5.*x* server. Then, click **Next**. On the Features page, specify a feature for the Web project. For example, enable **Default synchronization policy for CVS repository** to have a `.cvsignore` file generated for the `WEB-INF/classes` directory. Then, click **Next** or **Finish**.
   e. Click **Finish**.

4. Verify the contents of the new Web module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Web Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the Web module in a Navigator view.

Importing client applications

You can import an application client JAR file into a new or existing enterprise application using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **Application Client Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the application client JAR file and drop it on a view.
3. In the Import dialog, specify the application client file and project name:
   a. Click **App Client JAR file > Next**.
   b. Specify the application client JAR file to be imported. Use **Browse** to locate the JAR file and specify its full path name.
   c. Specify an application client project name. For example, if you are importing the `HelloWorld.jar` file, you might name the project `HelloWorld`. Click **New** and specify `HelloWorld` for the project name.
   d. Specify the enterprise archive (EAR) file into which to import the application client project.
   e. Click **Finish**.
4. Verify the contents of the new application client module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Application Client Modules** and view the new module.
   - Click **Window > Show View > Navigator** to see the associated files for the application client module in a Navigator view.

Importing EJB files

You can import an enterprise bean (EJB) JAR file and define a new enterprise archive (EAR) project and EJB module for the enterprise bean JAR file using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Click **File > Import**. Alternatively, you can right-click **EJB Modules** in a view such as the J2EE Hierarchy view and click **Import**. Or, on Windows platforms, you can drag the enterprise bean JAR file and drop it on a view.
3. In the Import dialog, specify the EJB JAR file and project name:
   a. Click **EJB JAR file > Next**.
   b. Specify the enterprise bean JAR file to import. Use **Browse** to locate the JAR file and specify its full path name.
   c. Specify an EJB project name. For example, if you are importing the `HelloWorld.jar` file, you might name the project `HelloWorld`. Click **New**, specify `HelloWorld` for the project name, specify whether you want to use the EJB 1.1 or 2.0 specification (EJB 2.0 is the default), and click **Next**.

d. Name the enterprise archive (EAR) file into which to import the enterprise bean JAR file. The name must be unique among EAR files in the directory.
e. Click **Finish**.
4. Verify the contents of the new EAR file and EJB module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Enterprise Applications** or **EJB Modules** and view the new modules.
   - Click **Window > Show View > Navigator** to see the associated files for the EAR file and EJB module in a Navigator view.

Importing RAR files or connectors

You can import a resource adapter archive (RAR) file, or connector, and define a new enterprise archive (EAR) project and connector module using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Right-click **Connector Modules** in a view such as the J2EE Hierarchy view and click **Import > Import Connector Module**.
3. In the Import dialog, specify the connector file and project name:
   a. Specify the name of the RAR file to import. Use **Browse** to locate the RAR file and specify its full path name.
   b. Specify a connector project name. For example, if you are importing the `HelloWorld.rar` file, you might name the project `HelloWorld`. Click **New**, specify `HelloWorld` for the project name, and click **Next**.
   c. If you want the connector project not to be part of an enterprise application, specify **Standalone connector project**.
   d. If the connector project is not to stand alone, name the enterprise archive (EAR) file into which to import the RAR file. The name must be unique among EAR files in the directory.
   e. If you want the Assembly Toolkit to overwrite existing resource files without first warning you that the files are changing, specify **Overwrite existing resources without warning**. The default is not to overwrite files without warning.
   f. Click **Finish**.
4. Verify the contents of the new connector module in either of the following ways:
   - In the J2EE Hierarchy view, expand **Enterprise Applications** or **Connector Modules** and view the new modules.
   - Click **Window > Show View > Navigator** to see the associated files for the connector module in a Navigator view.

Creating enterprise applications

Before you can deploy your archive files onto an application server, you must assemble them in an enterprise application archive (EAR) file. This article describes how to create a Java 2 Platform, Enterprise Edition (J2EE) enterprise application project using the Assembly Toolkit. After you create an enterprise application project, you can add (import) archive files such as Web application archives (WAR), resource adapter archives (RAR), enterprise bean (EJB) JAR files, and application client archives (JAR) files.

1. Start the Assembly Toolkit.

2. Click **File > New > Project**. Or, if you are working in the J2EE perspective, click **File > New > Enterprise Application Project** and skip step 3a below.
3. In the New Project dialog, create an enterprise application project:
   a. Click **J2EE > Enterprise Application Project > Next**.
   b. Specify whether you want an EAR file that supports J2EE 1.2 or 1.3, and click **Next**.
   c. On the Enterprise Application Project page, specify an EAR file name and location. To change the default project location, click **Browse** and specify a new location. Then, click **Next**.
   d. Optional: On the EAR Module Projects page, select the existing modules that you want to add to the new enterprise application project. To create new modules for this enterprise application, click **New Module**. On the New Module Project page, select **Create default module projects** to create modules for application client, enterprise bean (EJB), Web or connector projects. You can use the default project names for the modules or specify different project names. If you clear the **Create default module projects** check box, you can select a single module type and proceed with the proper wizard for that project type. Then, click **Finish** to create the project modules and add their names to the list of available modules on the EAR Module Projects page.
   e. Click **Finish**.
   f. Optional: Confirm that you want to view the J2EE Hierarchy view.
4. Verify the contents of the new enterprise application in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Application** and view the new EAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the enterprise application in a Navigator view.

Creating Web applications

In the Assembly Toolkit, you create and maintain resources for Web applications in Web projects. There are two types of Web projects, dynamic and static. Dynamic web projects can contain dynamic J2EE resources such as servlets, JavaServer Pages (JSP) files, filters, and associated metadata, in addition to static resources such as images and HTML files. Static Web projects only contain static resources.

Dynamic Web projects are always imbedded in enterprise application projects. Creating a Web project in the Assembly Toolkit requires that an enterprise application (EAR) project exist, or the Assembly Toolkit creates one for you. Creating a Web project updates the application.xml deployment descriptor of the specified enterprise application project to define the Web project as a module element. If you are importing a WAR file rather than creating a Web project new, the WAR Import wizard requires that you specify a Web project, which already requires an EAR project.

This article describes how to create a dynamic Web project using the Assembly Toolkit. For instructions on how to create a static Web project, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Web development > Tasks > Working with Web projects > Creating new static Web projects**.

1. Start the Assembly Toolkit.

2. Optional: View an animation file that shows how to create a dynamic Web project using the Assembly Toolkit. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Web development > Tasks > Working with Web projects > Creating new dynamic Web projects > Show Me**.
3. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
4. Click **File > New > Dynamic Web Project**.
5. On the Dynamic Web Project page of the New Web Project dialog:
   a. Specify a Web project (WAR file) name.
   b. Specify a location for the WAR file. To change the default WAR files location, click **Browse** and specify a new location.
   c. Decide whether you want to accept the defaults associated with a dynamic Web project or configure advanced options. If you want to accept the defaults, deselect **Configure advanced options**. Otherwise, select **Configure advanced options** and **Next**. Step 6 describes the defaults and advanced options for a dynamic Web project.
6. If you selected **Configure advanced options**, you can customize the Web project options:
   a. Specify a new or existing enterprise application (EAR) project to be associated with your new Web project for purposes of deployment. If you want to add a Web project as a module to another enterprise application project in the future, open the application.xml editor for the enterprise application project and select **Add** on the General page.
   b. Provide a **Context root** value. The context root is the Web application root, the top-level directory of your application when it is deployed to a Web server. The default value is the name of your Web project. You can change the context root after you create a project using the project Properties dialog, which you access from the project's context menu.
   c. From the J2EE Level drop-down list, select the appropriate Sun Microsystems Servlet and JSP specification level for the dynamic elements you plan to include in your Web project. Any new servlets and JSP files that you expect to create should adhere to the latest specification level available; previous specification levels are offered to accommodate any legacy dynamic elements that you expect to import into the project.
   d. Click **Next**.
   e. Optional: On the Features Page, select one or more of the Web project features and click **Next**.
   f. Optional: Select **Use a default Page Template for the Web Site** if you want your entire Web site to share a common page template. If you want to use one of the sample templates provided, select **Sample Template** and then choose one of the templates shown in the **Thumbnail** box. If you want to use a template of your own, select **User-defined Template** and then click **Browse** to select the template from the file system. The default is not to use a page template.
7. Click **Finish**. A new Web project is created, reflecting the J2EE folder structure that specifies the location of web content files, class files, class paths, the deployment descriptor, and supporting metadata.
8.
9. Verify the contents of the new Web project in either of the following ways:

- In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your Web project to view the new WAR file.
- Click **Window > Show View > Navigator** to see the associated files for the Web project in a Navigator view.

You can now begin creating or importing content for your Web project using the New File wizards or the Import wizards available from the **File** menu.

Creating application clients

Application client projects contain programs that run on networked client systems. An application client project is deployed as a JAR file.

In the Assembly Toolkit, you can create and add an application client project to a new or existing enterprise application project.

1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > Application Client Project**.
4. In the Application Client project creation dialog:
   a. Select the Java 2 Platform, Enterprise Edition (J2EE) specification version to which you want your project to adhere, and click **Next**.
   b. Name the application client project and specify its location. To change the default project location, click **Browse** and specify a new location. If you specify a non-default project location that is already being used by another project, the project creation will fail.
   c. Specify a new or existing enterprise application (EAR) project to be associated with your new application client project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project will be created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
   d. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project. This updates the runtime class-path and Java project build path with the appropriate JAR files. Application client modules, EJB modules, and Web modules can all have dependencies on EJB modules or utility JAR files. Modules cannot depend on WAR or application client JAR files.
   e. Click **Finish** to create the application client project.
5. Verify the contents of the new application client project in either of the following ways:
   - In the**J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your application client project to view the new JAR file.

- Click **Window > Show View > Navigator** to see the associated files for the application client project in a Navigator view.

After creating an application client project, you can edit the application client deployment descriptor if default properties are not sufficient. In the Client Deployment Descriptor editor, you can add enterprise bean, resource, or resource environment references as well as view and edit source code.

For detailed instructions on adding enterprise bean, resource, or resource environment references, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > J2EE application development > Tasks > Configuring application client modules with the client deployment descriptor editor**. Similar information is in the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click **WebSphere Studio Application Developer > J2EE development > Tasks > Configuring application client modules with the client deployment descriptor editor**.

Creating EJB modules

In the Assembly Toolkit, you can create and test enterprise beans that conform to the distributed component architecture defined in the Sun Microsystems Enterprise JavaBeans (EJB) specification and that support extended functionality for WebSphere Application Server.

You can create enterprise beans (either with or without inheritance) such as session beans, container-managed persistence (CMP) entity beans, bean-managed persistence (BMP) entity beans, or message-driven beans. Using the EJB deployment descriptor editor of the Assembly Toolkit, you can set deployment descriptor and assembly properties for enterprise beans.

This article describes how to create an EJB project (or EJB module) using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > EJB Project**.
4. In the New EJB Project dialog:
   a. Select the EJB specification version to which you want your EJB project to adhere, and click **Next**. If you plan on using EJB 2.0 enterprise beans, you must specify an EJB 2.0 project. You can add EJB 1.1 enterprise beans to an EJB 2.0 project. An EJB 2.0 project must exist in a J2EE 1.3 enterprise application project. Your available options can differ, depending on the J2EE preferences defined.
   b. Name the EJB project and specify its location. To change the default project location, click **Browse** and specify a new location. If you specify a non-default project location that is already being used by another project, the project creation will fail.
   c. Specify a new or existing enterprise application (EAR) project to be associated with your new EJB project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project will be created in the default location with the lowest compatible J2EE version

based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.

d. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project. Note that this page is available only if you are using an existing enterprise application project.

e. Click **Finish** to create the EJB project.

5. Verify the contents of the new EJB project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your EJB project to view the new JAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the EJB project in a Navigator view.

After you have an EJB project to hold enterprise beans, you can do the following:
- Create or import enterprise beans to your EJB project.
- Add methods to the home and remote interfaces.
- Add custom finders.
- Add and define additional CMP fields.
- Add relationships.
- Edit the EJB deployment descriptor if default properties are not sufficient.
- Create EJB access beans and use them to create your client application.
- Map enterprise beans to RDB tables.

For detailed instructions on creating CMP fields or CMP finder methods for entity beans, relating CMP fields, adding methods to interfaces, or managing enterprise beans, see the Assembly Toolkit online help. In the Assembly Toolkit, click **Help > Help Contents > Assembly Toolkit information > Enterprise JavaBeans (EJB) development > Tasks**. Similar information is in the IBM WebSphere Studio Application Developer InfoCenter at `http://publib.boulder.ibm.com/infocenter/wsphelp/index.jsp`. Click **WebSphere Studio Application Developer > J2EE development > Tasks > Developing EJB applications**.

Creating connector modules

A *connector* is a J2EE component that provides access to Enterprise Information Systems (EIS), and must comply to the J2EE Connector architecture (JCA). An *Enterprise Information System (EIS)* is a set of related classes that lets an application access a resource such as data, or an application on a remote server, often called a resource adapter.

This article describes how to create a connector project using the Assembly Toolkit.

1. Start the Assembly Toolkit.
2. Open the J2EE perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
3. Click **File > New > Connector Project**.
4. In the New Connector Project dialog:

a. If you want your connector project to be a stand-alone project, select **Standalone connector project**. Selecting that the connector project stand alone disables the enterprise archive (EAR) project option at the bottom of the page.
b. Name the connector project and specify its location. To change the default project location, click **Browse** and specify a new location.
c. If you specified that the connector project stand alone, specify a new or existing enterprise application (EAR) project to be associated with your new connector project for purposes of deployment. Select an existing enterprise application project from the drop-down list or type a new project name. Or, click **New** and create a new enterprise application. Note that if you type a new EAR project name, the EAR project will be created in the default location with the lowest compatible J2EE version based on the version of the project being created. If you want to specify a different version or a different location for the enterprise application, you must click **New** and create a new enterprise application.
d. Optional: If you are creating a new enterprise application project or if you have no module dependencies to specify, skip this step. Otherwise, click **Next** to specify module and JAR file dependencies. On the Module Dependencies page, select dependent JAR files or modules within the associated enterprise application project.
e. Click **Finish** to create the connector project.

5. Verify the contents of the new connector project in either of the following ways:
   - In the **J2EE Hierarchy** view, expand **Enterprise Applications** and the enterprise application associated with your connector project to view the new JAR file.
   - Click **Window > Show View > Navigator** to see the associated files for the connector project in a Navigator view.

Editing deployment descriptors

A deployment descriptor is an extensible markup language (XML) file that describes how to deploy a module or application by specifying configuration and container options. When you create a module, the Assembly Toolkit creates deployment descriptor files for the module.

You can edit a deployment descriptor file manually. However, it is preferable to edit a deployment descriptor using an Assembly Toolkit deployment descriptor editor to ensure that the deployment descriptor has valid properties and that its references contain appropriate values.

Deployment descriptor editor	Resources modified in the editor
Application deployment descriptor editor	• `application.xml` • `ibm-application-bnd.xmi` • `ibm-application-ext.xmi`
Web deployment descriptor editor	• `WEB-INF/web.xml` • Binding information • IBM binding and extensions information such as `ibm-web-bnd.xmi` and `ibm-web-ext.xmi` files
Enterprise bean (EJB) deployment descriptor editor	• `ejb-jar.xml` • `ibm-ejb-jar-bnd.xml` • `ibm-ejb-jar-ext.xml` • `ibm-ejb-access-bean.xml`

Client deployment descriptor editor	• `application-client.xml` • `ibm-application-client-bnd.xmi` • `ibm-application-client-ext.xmi`
Web services editor	• `webservices.xml` • `ibm-webservices-bnd.xmi` • `ibm-webservices-ext.xmi`
Web services client editor	• `webservicesclient.xml` • `ibm-webservicesclient-bnd.xmi` • `ibm-webservicesclient-ext.xmi`

1. Ensure that you are working in the J2EE Perspective. Click **Window > Open Perspective > J2EE**.
2. In a J2EE Hierarchy view (**Window > Show View > J2EE Hierarchy**), right-click the module with deployment descriptor values that you want to browse or edit, and click **Open With > Deployment Descriptor Editor**. A deployment descriptor editor for the module displays in a view. You can click tabs such as **Overview**, **Module**, **Security**, and **Source** at the bottom of the view to browse or edit specific deployment descriptor values. Clicking **Source** displays editable source code; it is preferable to edit values in fields on or accessible from the other tabs rather than edit the source code manually.
3. Edit the deployment descriptor values as desired. For information on fields in the deployment descriptor editor, press F1 and click a topic.
4. Save your changes to the deployment descriptor.
   a. Close the deployment descriptor editor.
   b. When prompted, click **Yes** to indicate that you want to save changes to the deployment descriptor.

   You also can save changes to deployment descriptors at any time by pressing Ctrl-S.

Mapping enterprise beans to database tables

You can map enterprise bean JAR files (EJB modules) to relational database (RDB) tables using the EJB to RDB Mapping wizard of the Assembly Toolkit. The wizard creates EJB to RDB mappings for the following situations:

Existing enterprise bean but no database schema
: *Top Down* mapping generates a default database schema and a mapping from one or more existing enterprise beans.

Existing database schema but no enterprise bean
: *Bottom Up* mapping generates one or more enterprise beans and mappings from an existing database schema.

Existing enterprise bean and database schema
: *Meet In the Middle* mapping matches existing enterprise beans with existing database tables. You can match by name, by name and type, or by neither.

1. In the J2EE Hierarchy view, right-click the EJB module.
2. Click **Generate > EJB to RDB Mapping**.
3. After the wizard opens, press F1 and select a type of mapping. The online help provides detailed information on generating a mapping.
4. For EJB 2.0 projects, on the EJB to RGB Mapping page specify whether you want to create a new backend (*Top Down*) or use an existing backend (*Bottom Up* or *Meet In the Middle*) where the schema exists in the backend but without a mapping file. If you previously generated a mapping, you can create and map unmapped elements or open the mapping editor to manually make changes. In

EJB 2.0, your mapping and schema files make up a *backend* for EJB 2.0 projects. You can have multiple backend folders for each project; for example, one DB2 and one Oracle backend. The wizard uses one database backend only as the default, but you can define as many as you need.

5. Follow the instructions in the wizard and in the online help.
6. Click **Finish** to generate the mapping.

Verifying archive files

The Assembly Toolkit validates code when you request code validation manually, automatically during a resource change, and automatically during a build.

As part of validating the code, the validation checks for the following:

- Required deployment properties contain values.
- Values specified for environment entries match their associated Java types.
- In both enterprise archive (EAR) and Web archive (WAR) files:
  - The target enterprise bean of the link exists for enterprise bean (EJB) references.
  - The target role exists for security role references.
  - Security roles are unique.
- Each module listed in the deployment descriptor exists in the archive for EAR files.
- Files for icons, servlets, error and welcome pages listed in the deployment descriptor have corresponding files in the archive for WAR files.
- For EJB modules:
  - All class files referenced in the deployment descriptor exist in the JAR file.
  - Method signatures for enterprise bean home, remote and implementation classes are compliant with the EJB 2.0 specification.

1. Optional: Specify whether you want automatic code validation during a resource change or during a build. The default is for automatic code validation.
   a. In the J2EE Hierarchy view, right-click on a project.
   b. Click **Properties > Validation**.
   c. Ensure that the **Run validation** options for builds and for automatic validation are selected. Select **Override validation preferences** to disable automatic code validation.
   d. If you changed the **Validation** settings, click **Apply** or **OK**.
2. Optional: Specify validation options for a project. The default is to check all validators for a project during code validation. For an enterprise application project, the validators might be for DTD, EAR, Web services, XML, XML schema, or XSL files.
   a. In the J2EE Hierarchy view, right-click the project containing the code that you want to validate.
   b. Click **Properties > Validation**.
   c. Select **Override validation preferences**.
   d. Select the validators you want checked during code validation.
   e. If you changed the **Validation** settings, click **Apply** or **OK**.
3. Right-click the project containing the code that you want to validate and click **Run Validation** to manually validate the code.

The results of the code validation are shown in a Tasks view. For information on the results, select an entry in the Tasks view, press F1, and click **Tasks view**.

If your application module contains EJB modules, generate code for deployment. Otherwise, you are ready to deploy the application module (or stand-alone Web module) onto the application server.

Generating code for EJB deployment

This task assumes you have already assembled an enterprise bean (EJB) module, added it to an application, saved the application, and verified the application.

Before installing your application in WebSphere Application Server, you must generate deployment code for the application. This step is required for EJB modules and for any enterprise application archive (EAR) files that contain EJB modules. During code generation, the Assembly Toolkit prepares entity bean (JAR) files for deployment in a run-time environment. If your EJB project contains container-managed persistence (CMP) beans that have not been mapped, generating deployment code creates a default top-down mapping.

1. If you have turned automatic validation off, manually validate your enterprise beans before generating deployment code for them. If validating your beans results in compilation errors or validation errors, fix the errors before generating deployment code. However, if validating your beans results in warning or information messages, you can generate deployment code.
2. If you have changed the class path of your EJB project, ensure that the source folder for your EJB project is at the beginning of the class path of the project. Generating deployment code imports both the JAR file and the source code of the JAR file, so entries on the class path must be in the correct order.
3. In the J2EE Hierarchy view of the Assembly Toolkit, right-click on the enterprise application (EAR file) or EJB module (enterprise bean JAR file) for which you want to generate code for deployment.
4. Click a **Generate Deployment Code** option.
   - For EAR files, click **Generate Deployment Code**.
   - For enterprise bean JAR files, click **Generate > Deployment and RMIC Code > *EJB_module* > Finish**.

   Alternatively, you can generate deployment code for enterprise bean JAR files using the deployment tool for Enterprise JavaBeans (ejbdeploy) from a command prompt. For a detailed list of available options in the EJBDeploy tool, enter `ejbdeploy` from a command prompt.

   For the z/OS environment: Use the `-e` option to display console output in ASCII characters rather than EBCDIC characters. Thus, enter `./ejbdeploy.sh -e` to see a list of available options and use the syntax `./ejbdeploy.sh -e <input> <working> <output>` and so on.

Code is generated into the folder where your enterprise beans are located. Problems with the generation of Java RMI stub compiler (RMIC) code result in a window that displays error messages.

Install the Java 2 Platform, Enterprise Edition (J2EE) application on your server machine. You can install the application onto a server using the administrative console. Before installing the application, you might need to set class paths.

Generating code for Web service deployment

This task assumes you have already assembled a module enabled with Web services, added it to an application, saved the application, and verified the application.

Before installing your application in WebSphere Application Server, you must generate deployment code for the application. This step is required for Web services-enabled modules and for any enterprise application archive (EAR) files that contain Web services-enabled modules.

1. If you have turned automatic validation off, manually validate any modules that use Web services with the JSR109 Web services validator before generating deployment code for them. If validating your module results in compilation errors or validation errors, fix the errors before generating deployment code. However, if validating your module results in warning or information messages, you can generate deployment code.
2. In the J2EE Hierarchy view of the Assembly Toolkit, right-click on the Web services-enabled module (WAR, enterprise bean JAR, or application client JAR file) for which you want to generate code for deployment.
3. Click **Web Services > Deploy Web Service**. Alternatively, you can generate deployment code for Web services-enabled modules using the deployment tool for Web services (wsdeploy) from a command prompt.
4. If messages indicate that automatic file overwriting is not enabled, click **Yes to All** so the generated files are added to the module.
5. If errors such as *Unbound classpath variable: WAS_50_PLUGINDIR* appear in the Tasks list, change the Java build path libraries properties to define that variable to be the WebSphere Application Server installation directory.

Install the Java 2 Platform, Enterprise Edition (J2EE) application on your server machine. You can install the application onto a server using the administrative console. Before installing the application, you might need to set class paths.

Assembly Toolkit: Resources for learning

Use the following links to find relevant supplemental information about the Assembly Toolkit. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- Programming instructions and examples
- Programming specifications
- Administration

Programming instructions and examples

- Developing and testing a complete J2EE "Hello World" application with WebSphere Studio V5

- Getting to know WebSphere Studio Application Developer: Its capabilities, technologies, and relationship to the open-source Eclipse IDE
- Developing and Deploying an End-to-end J2EE Application to JBoss Application Server using WebSphere Studio V5
- JMS Applications with WebSphere Studio V5 -- Part 1: Developing a JMS Point-to-Point Application
- WebSphere Studio Version 5 Tips and Techniques
- Java 2 Enterprise Edition: Books index

Programming specifications
- J2EE 1.3 specification
- EJB specifications
- Servlet specifications

Administration
- Application Client files
- Connector RAR files

Chapter 18. Deploying and managing applications

After you develop an enterprise application and configure an application server, you can use the administrative console to install application files on the server and manage the activity of deployed applications.

1. Install your application on your application server.
2. Start and stop applications.
3. Edit the administrative configuration for an application. Go to the settings page for an application, change the values for settings as needed, and click **OK**.
4. Export applications.
5. Export DDL files.
6. Update application binary files.
7. Uninstall applications.

After making changes to administrative configurations of your applications, ensure that you click **Save** on the administrative console taskbar to save the changes.

Enterprise applications

Enterprise applications (or J2EE applications) are applications that conform to the Java 2 Platform, Enterprise Edition, specification.

Enterprise applications can consist of the following:
- Zero or more EJB modules
- Zero or more Web modules
- Zero or more connector modules (packaged in RAR files)
- Zero or more application client modules
- Additional JAR files containing dependent classes or other components required by the application
- Any combination of the above

A J2EE application is represented by, and packaged in, an enterprise archive (EAR) file.

Installing a new application

To install an enterprise application to a WebSphere Application Server configuration, you can use the administrative console or the wsadmin tool. The steps below describe how to use the administrative console to install an application, EJB component, or Web module.

Note: Once you start performing the steps below, click **Cancel** to exit if you decide not to install the application. Do not simply move to another administrative console page without first clicking **Cancel** on an application installation page.

1. Click **Applications > Install New Application** in the console navigation tree. The first of two Preparing for application install pages is shown.
2. On the first Preparing for application install page:
   a. Specify the full path name of the source application file (.ear file otherwise known as an EAR file). The EAR file that you are installing can be either on the client machine (the machine that runs the Web browser) or on the

© Copyright IBM Corp. 2003

server machine (the machine to which the client is connected). If you specify an EAR file on the client machine, then the administrative console uploads the EAR file to the machine on which the console is running and proceeds with application installation. You can also specify a stand-alone WAR or JAR file for installation.

   b. If you are installing a stand-alone WAR file, specify the context root.
   c. Click **Next**.
3. On the second Preparing for application install page:
   a. Select whether to generate default bindings. Using the default bindings causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not altered. You can customize default values used in generating default bindings. For example, you can specify JNDI prefix for all the EJB files in EJB modules, default data source and connection factory settings for EJB modules, virtual host for web modules, and so on. ″"Preparing for application installation settings" on page 904″ describes available customizations and provides sample bindings.
   b. Click **Next**. The Install New Application pages are now shown. If you chose to generate default bindings, you can proceed to the Summary step (step 23 below). ″"Example: Installing an EAR file using the default bindings" on page 908″ provides sample steps.
4. On the **Step: Provide options to perform the installation** panel, provide values for the following settings specific to WebSphere Application Server. Default values are used if you do not specify a value.
   a. For **Pre-compile JSP**, specify whether to precompile JSP files as a part of installation. The default is not to precompile JSP files.
   b. For **Directory to Install Application**, specify the directory to which the application EAR file will be installed. The default value is the value of APP_INSTALL_ROOT/*cell_name*, where the APP_INSTALL_ROOT variable is *install_root*/installedApps; for example, C:\WebSphere\AppServer\installedApps*cell_name*.

      Note: If an installation directory is not specified when an application is installed on a single-server (base) configuration, the application is installed in APP_INSTALL_ROOT/*base_cell_name*. When the base server is made a part of a Network Deployment configuration (using the addNode utility), the cell name of the new configuration becomes the cell name of the deployment manager node. If the `-includeapps` option is used for the addNode utility, then the applications that are installed prior to the addNode operation still use the installation directory APP_INSTALL_ROOT/*base_cell_name*. However, an application that is installed after the base server is added to the network configuration uses the default installation directory APP_INSTALL_ROOT/*network_cell_name*. To move the application to the APP_INSTALL_ROOT/*network_cell_name* location upon running the addNode operation, you should explicitly specify the installation directory as `${APP_INSTALL_ROOT}/${CELL}` during installation. In such a case, the application files can always be found under APP_INSTALL_ROOT/*current_cell_name*.
   c. For **Distribute Application**, specify whether WebSphere Application Server expands or deletes application binaries in the installation destination. The default is to enable application distribution. As a result, when you save changes in the console, application binaries for newly installed applications are expanded to the directory specified. The binaries are also deleted when you uninstall and save changes to the configuration. If you disable this

option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application is expected to run.

d. For **Use Binary Configuration**, specify whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the deployment.xml file (default), or those located in the EAR file. The default is not to use the binary configuration.

e. For **Deploy EJBs**, specify whether the EJBDeploy tool runs during application installation. The tool generates code needed to run EJB files. The default is not to run the EJBDeploy tool. You must enable this setting if the EAR file was assembled using theApplication Assembly (AAT) toolorAssembly Toolkit and the EJBDeploy tool was not run during assembly, if the EAR file was not assembled using theAATorAssembly Toolkit tool, or if the EAR file was assembled using versions of the Application Assembly Tool (AAT) previous to Version 5. Note that enabling this setting might cause the installation program to run for several minutes.

f. For **Application Name**, name the application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.

g. For **Create MBeans for Resources**, specify whether to create MBeans for various resources (such as servlets or JSP files) within an application when the application is started. The default is to create MBean instances.

h. For **Enable class reloading**, specify whether to enable class reloading when application files are updated. The default is not to enable class reloading.

i. For **Reload Interval**, specify the number of seconds to scan the application's file system for updated files. The default is the value of the reload interval attribute in the IBM extension (META-INF/ibm-application-ext.xmi) file of the EAR file. This setting takes effect only if class reloading is enabled.

   The reload interval specified here overrides the value specified in the IBM extensions for each Web module in the EAR file (which in turn overrides the reload interval specified in the IBM extensions for the application in the EAR file).

j. 5.0.2 + For **Deploy WebServices**, specify whether the Web services deploy tool `wsdeploy` runs during application installation. The tool generates code needed to run applications using Web services. The default is not to run the `wsdeploy` tool. You must enable this setting if the EAR file contains modules using Web services and has not previously had the `wsdeploy` tool run on it, either from the **Web Services > Deploy Web Service** menu of the Assembly Toolkit or from a command line.

5. If your application uses EJB modules, on the **Step: Provide JNDI Names for Beans** panel, specify a JNDI name for each enterprise bean in every EJB module. You must specify a JNDI name for every enterprise bean defined in the application. For example, for the EJB module MyBean.jar, specify `MyBean`.

6. If your application uses EJB modules that contain Container Managed Persistence (CMP) beans that are based on the EJB 1.x specification, for **Step: Provide default datasource mapping for modules containing 1.x entity beans**, specify a JNDI name for the default data source for the EJB modules. The default data source for the EJB modules is optional if data sources are specified for individual CMP beans.

7. If your application has CMP beans that are based on the EJB 1.x specification, for **Step: Map datasources for all 1.x CMP**, specify a JNDI name for data sources to be used for each of the 1.x CMP beans. The data source attribute is optional for individual CMP beans if a default data source is specified for the EJB module that contains CMP beans. If neither a default data source for the EJB module nor a data source for individual CMP beans are specified, then a validation error displays after you click **Finish** (step 23) and the installation is cancelled.
8. If your application defines EJB references, for **Step: Map EJB references to beans**, specify JNDI names for enterprise beans that represent the logical names specified in EJB references. Each EJB reference defined in the application must be bound to an EJB file before clicking **Finish** on the Summary panel.
9. If your application defines resource references, for **Step: Map resource references to resources**, specify JNDI names for the resources that represent the logical names defined in resource references. Each resource reference defined in the application must be bound to a resource defined in your WebSphere Application Server configuration before clicking on **Finish** on the Summary panel.
10. If your application uses Web modules, for **Step: Map virtual hosts for web modules**, select a virtual host from the list that should map to a Web module defined in the application. The port number specified in the virtual host definition is used in the URL that is used to access artifacts such as servlets and JSP files in the Web module. Each Web module must have a virtual host to which it maps. Not specifying all needed virtual hosts will result in a validation error displaying after you click **Finish** on the Summary panel.
11. On the **Step: Map modules to application servers** panel, for every module select a target server or a cluster from the **Clusters and Servers** list. Select the check box beside **Module** to select all of the application modules or select individual modules.
12. If the application has security roles defined in its deployment descriptor then, for **Step: Map security roles to users/groups**, specify users and groups that are mapped to each of the security roles. Select the check box beside **Role** to select all of the roles or select individual roles. For each role, you can specify if predefined users such as **Everyone** or **All Authenticated users** are mapped to it. To select specific users or groups from the user registry:
    a. Select a role and click **Lookup users** or **Lookup groups**.
    b. On the Lookup users/groups panel shown, enter search criteria to extract a list of users or groups from the user registry.
    c. Select individual users or groups from the results displayed.
    d. Click **OK** to map the selected users or groups to the role selected on the **Step: Map security roles to users/groups** panel.
13. If the application has Run As roles defined in its deployment descriptor, for **Step: Map RunAs roles to user**, specify the Run As user name and password for every Run As role. Run As roles are used by enterprise beans that must run as a particular role while interacting with another enterprise bean. Select the check box beside **Role** to select all of the roles or select individual roles. After selecting a role, enter values for the user name, password, and verify password and click **Apply**.
14. If your application contains EJB 1.x CMP beans that do not have method permissions defined for some of the EJB methods, for **Step: Ensure all**

unprotected 1.x methods have the correct level of protection, specify if you want to leave such methods unprotected or assign protection with deny all access.

15. If your application contains message driven enterprise beans, for **Step: Provide Listener Ports for messaging beans**, provide a listener port name for every message driven bean. If a name is not specified for each bean, then a validation error displays after you click on **Finish** on the Summary panel.
16. If your application uses EJB modules that contain CMP beans that are based on the EJB 2.0 specification, for **Step: Provide default datasource mapping for modules containing 2.0 entity beans**, specify a JNDI name for the default data source and the type of resource authorization to be used for the default data source for the EJB modules. The default data source for EJB modules is optional if data sources are specified for individual CMP beans.
17. If your application has CMP beans that are based on the EJB 2.0 specification, on the **Step: Map datasources for all 2.0 CMP** panel, for each of the 2.0 CMP beans specify a JNDI name and the type of resource authorization for data sources to be used. The data source attribute is optional for individual CMP beans if a default data source is specified for the EJB module that contains CMP beans. If neither a default data source for the EJB module nor a data source for individual CMP beans are specified, then a validation error is shown after you click **Finish** and installation is cancelled.
18. If your application contains EJB 2.0 CMP beans that do not have method permissions defined in the deployment descriptors for some of the EJB methods, on the **Step: Ensure all unprotected 2.0 methods have the correct level of protection** panel, specify whether you want to assign a specific role to the unprotected methods, add the methods to the exclude list, or mark them as unchecked. Methods added to the exclude list are marked as uncallable. For methods marked unchecked no authorization check is performed prior to their invocation.
19. If the **Deploy EJBs** setting is enabled on the **Provide options to perform the installation** panel, then you can specify options for the EJBDeploy tool on the **Step: Provide options to perform the EJB Deploy** panel. On this panel, you can specify extra class path, rmic options, database types, and database schema names to be used while running the EJBDeploy tool. The tool is run on the EAR file during installation after you click **Finish**.
20. If your application contains resource environment references, for **Step: Mapping Resource Environment References to Resources**, specify JNDI names of resources that map to the logical names defined in resource environment references. If each resource environment reference does not have a resource associated with it, a validation error is shown after you click **Finish**.
21. If your application defines **Run-As Identity** as *System Identity*, for **Step: Replacing RunAs System to RunAs Roles**, you can optionally change it to *Run-As role* and specify a user name and password for the Run As role specified. Selecting *System Identity* implies that the invocation is done using the WebSphere Application Server security server ID and should be used with caution as this ID has more privileges.
22. If your application has resource references that map to resources that have an Oracle database doing backend processing, for **Step: Specify the isolation level for Oracle type provider**, specify or correct the isolation level to be used for such resources when used by the application. Oracle databases support ReadCommitted and Serializable isolation levels only.

23. On the Summary panel, verify the cell, node, and server onto which the application modules will install. Beside the **Cell/Node/Server** option, click **Click here** and verify the settings. Then click **Finish**.

    Note: After clicking **Finish**, if you receive an OutOfMemory exception and the source application file does not install, your system might not have enough memory or your application might have too many modules in it to install successfully onto the server. If lack of system memory is not the cause of the exception, package your application again so the .ear file has fewer modules. If lack of system memory and the number of modules are not the cause of the exception, check the options you specified on the Java Virtual Machine page of the application server running the administrative console. Then, try installing the application file again.
24. Associate any shared libraries that the application needs to the application.
25. Click **Save** on the administrative console taskbar to save the changes to your configuration. The application is registered with the administrative configuration and application files are copied to the target directory, which is *install_root*/installedApps/*cell_name* by default or the directory that you designate. For the single-server (base) installation, application files are copied to the destination directory when you click **Save**; for the Network Deployment installation, files are copied to remote nodes when the configuration on the deployment manager synchronizes with the configuration on individual nodes.
26. Start the application.
27. Test the application. For example, point a Web browser at the URL for the deployed application and examine the performance of the application. If necessary, update the application.

Preparing for application installation settings

Use this page to install an application (EAR file) or module (JAR or WAR file).

To view this administrative console page, click **Applications > Install New Application**.

Follow the steps on this page to install an application or module. You must complete, at minimum, the first step; you must complete some or all of the later steps, depending on whether you are installing an application, EJB module, or Web module.

Path

Specifies the fully qualified path to the .ear, .jar, or .war file for the enterprise application.
Use **Local path** if the browser and application files are on the same machine (whether or not the server is on that machine, too).

Use **Server path** if the application file resides on any node in the current cell context. Only .ear, .jar, or .war files are shown during the browsing.

During application installation, application files are typically uploaded from a client machine running the browser to the server machine running the administrative console, where they are deployed. In such cases, the Web browser running the administrative console is used to select EAR, WAR, or JAR modules to upload to the server machine.

In some cases, however, the application files reside on the file system of any of the nodes in a cell. To have the application server install these files, use the **Server path** option.

You can also use this option to specify an application file already residing on the machine running the application server. For example, the field value on Windows NT might be `C:\WebSphere\AppServer\installableApps\test.ear`. If you are installing a stand-alone WAR module, then you also must specify the context root.

Context Root

Specifies the context root of the Web application (WAR).
This field is used only to install a stand-alone WAR file. The context root is combined with the defined servlet mapping (from the WAR file) to compose the full URL that users type to access the servlet. For example, if the context root is /gettingstarted and the servlet mapping is MySession, then the URL is `http://host:port/gettingstarted/MySession`.

Generate Default Bindings

Specifies whether to generate default bindings. If you place a check mark in the check box, then any incomplete bindings in the application are filled in with default values. Existing bindings are not altered.
By choosing this option, you can directly jump to the Summary step and install the application if none of the steps have a red asterisk (*) next to them. A red asterisk denotes that the step has incomplete data and requires a valid value. On the Summary panel, verify the cell, node and server on which the application is installed.

Bindings are generated as follows:

- EJB JNDI names are generated of the form *prefix/ejb-name*. The default prefix is `ejb`, but can be overridden. The *ejb-name* is as specified in the deployment descriptors `<ejb-name>` tag.
- EJB references are bound as follows: If an <ejb-link> is found, it is honored. Otherwise, if a unique enterprise bean is found with a matching home (or local home) interface as the referenced bean, the reference is resolved automatically.
- Resource reference bindings are derived from the <res-ref-name> tag. Note that this action assumes that the java:comp/env name is the same as the resource global JNDI name.
- Connection factory bindings (for EJB 2.0 JAR files) are generated based on the JNDI name and authorization information provided. This action results in default connection factory settings for each EJB 2.0 JAR file in the application being installed. No bean-level connection factory bindings are generated.
- Data source bindings (for EJB 1.1 JAR files) are generated based on the JNDI name, data source user name password options. This results in default data source settings for each EJB JAR file. No bean-level data source bindings are generated.
- Message-driven bean (MDB) listener ports are derived from the MDB <ejb-name> tag with the string `Port` appended.
- For .war files, the virtual host is set as `default_host` unless otherwise specified.

The default strategy suffices for most applications or at least for most bindings in most applications. However, it does not work if:

- You want to explicitly control the global JNDI names of one or more EJB files.
- You need tighter control of data source bindings for CMPs. That is, you have multiple data sources and need more than one global data source.
- You must map resource references to global resource JNDI names that are different from the java:comp/env name.

In such cases, you can alter the behavior with an XML document (a custom strategy). Use the **Specific bindings file** field to specify a custom strategy and see the field's help for examples.

Prefixes

Specifies prefixes to use for generated JNDI names.

Override

Specifies whether to override existing bindings.
If this check box is checked, the existing bindings are overridden by the generated ones.

EJB 1.1 CMP bindings

Specifies the default data source JNDI name.
If the **Default Bindings for EJB 1.1 CMPs** radio button is selected, specify the JNDI name for the default data source to be used with the CMP 1.1 beans. Also specify the user ID and password for this default data source.

Connection Factory Bindings

Specifies the default data source JNDI name.
If the **Default connection factory bindings** radio button is selected, specify the JNDI name for the default data source to be used with the bindings. Also specify the resource authorization.

Virtual Host

Specifies the virtual host for WAR modules.

Specific bindings file

Specifies a bindings file that overrides the default binding.
Alter the behavior of the default binding with an XML document (*aka* custom strategy). Custom strategies extend the default strategy so you only need to customize those areas where the default strategy is insufficient. That is, you only need to describe how you want to change the bindings generated by the default strategy; you do not have to define bindings for the entire application.

Brief examples of how to override various aspects of the default bindings generator follow:

Controlling an EJB JNDI name

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>helloEjb.jar</jar-name>
<!-- this name must match the module name in the .ear file -->
      <ejb-bindings>
        <ejb-binding>
         <ejb-name>HelloEjb</ejb-name>
<!-- this must match the <ejb-name> entry in the EJB jar DD -->
         <jndi-name>com/acme/ejb/HelloHome</jndi-name>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Setting the connection factory binding for an EJB JAR file

```
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <connection-factory>
        <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
        <res-auth>Container</res-auth>
      </connection-factory>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Setting the connection factory binding for an EJB file

```
<?xml version="1.0">
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>yourEjb20.jar</jar-name>
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourCmp20</ejb-name>
<!-- this matches the ejb-name tag in the DD -->
          <connection-factory>
           <jndi-name>eis/jdbc/YourData_CMP</jndi-name>
           <res-auth>PerConnFact</res-auth>
          </connection-factory>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
 </module-bindings>
</dfltbndngs>
```

Overriding a Resource Ref Binding from a WAR, EJB JAR file, or J2EE client JAR file

Example code for overriding a Resource Ref Binding from a WAR file follows. Use similar code to override a Resource Ref Binding from an enterprise bean (EJB) JAR file or a J2EE client JAR file.

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <war-binding>
      <jar-name>hello.war</jar-name>
      <resource-ref-bindings>
        <resource-ref-binding>
          <!-- the following must match the resource-ref in the DD -->
          <resource-ref-name>jdbc/MyDataSrc</resource-ref-name>
          <jndi-name>war/override/dataSource</jndi-name>
        </resource-ref-binding>
      </resource-ref-bindings>
    </war-binding>
  </module-bindings>
</dfltbndngs>
```

Overriding MDB JMS listener ports

```
<?xml version="1.0"?>
<!DOCTYPE dfltbndngs SYSTEM "dfltbndngs.dtd">
<dfltbndngs>
  <module-bindings>
    <ejb-jar-binding>
      <jar-name>YourEjbJar.jar</jar-name>
```

```
      <ejb-bindings>
        <ejb-binding>
          <ejb-name>YourMDB</ejb-name>
          <listener-port>yourMdbListPort</listener-port>
        </ejb-binding>
      </ejb-bindings>
    </ejb-jar-binding>
  </module-bindings>
</dfltbndngs>
```

Example: Installing an EAR file using the default bindings

An example of a simple .ear file installation using the default bindings follows:

1. Go to the Preparing for application install pages. Click **Applications > Install an Application** in the console navigation tree.
2. For **Path**, specify the full path name of the .ear file. For this example, the base file name is my_appl.ear and the file resides on a server at C:\sample_apps.
   a. Select the **Server path** radio button and click **Browse**.
   b. On the Browse Remote Filesystems page, click on the name of the node that holds the my_appl.ear file, **C:**, **sample_apps**, **my_appl.ear**, and then **OK**.
3. Now that a value is given for **Path**, on the first Preparing for application install page, click **Next**.
4. On the second Preparing for application install page, place a checkmark beside the **Generate Default Bindings** check box and click **Next**. Using the default bindings causes any incomplete bindings in the application to be filled in with default values. Existing bindings are not changed. By choosing this option, you can directly jump to the Summary step.
5. On the Install New Application page, click on **Summary**, the last step.
6. On the Summary panel, verify the cell, node, and server onto which the application files will install.
   a. Beside the **Cell/Node/Server** option, click **Click here**.
   b. On the **Map modules to application servers** panel, select the server onto which the application files will install from the **Clusters and Servers** list, place a checkmark in the check box beside **Module** to select all of the application modules, and click **Next**.

   Because my_appl.ear does not require any additional settings to complete an installation, the Summary panel displays again.
7. On the Summary panel, click **Finish**.

Enterprise application collection

Use this page to view and manage enterprise applications.

To view this administrative console page, click **Applications > Enterprise Applications**.

Name

Specifies the name of the installed (or deployed) application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.

Status

Indicates whether the application deployed on the application server is started, stopped, or unavailable.

Enterprise application settings

Use this page to configure an enterprise application.

To view this administrative console page, click **Applications > Enterprise Applications >***application_name*.

Name

Specifies a logical name for the application. Application names must be unique within a cell and cannot contain characters that are not allowed in object names.

Data type	String

Starting Weight

Specifies the order in which applications are started when the server starts. The application with the lowest starting weight is started first.

Data type	Integer
Default	1
Range	0 to 2147483647

Application Binaries

Specifies the directory to which the application EAR file will be installed. The default value is the value of APP_INSTALL_ROOT/*cell_name*, where the APP_INSTALL_ROOT variable is *install_root*/installedApps; for example, C:\WebSphere\AppServer\installedApps*cell_name*.
You can specify an absolute path or use a pathmap variable such as ${MY_APPS}. You can use a pathmap variable in any installation though it is particularly needed when installing an application on a cluster with members on heterogeneous nodes because, in such cases, there might not be a single way to specify an absolute path. A WebSphere Application Server variable ${CELL} that denotes the current cell name can also be in the pathmap variable; for example, ${MY_APP}/${CELL}.

Data type	String
Units	Full path name

Use Metadata From Binaries

Specifies whether the application server uses the binding, extensions, and deployment descriptors located with the application deployment document, the `deployment.xml` file (default), or those located in the enterprise application resource (EAR) file.

Data type	Boolean
Default	false

Enable Distribution

Specifies whether WebSphere Application Server expands or deletes application binaries in the installation destination. The default is to enable application distribution. Application binaries for installed applications are expanded to the directory specified. The binaries are also deleted when you uninstall and save changes to the configuration. If you disable this option, then you must ensure that the application binaries are expanded appropriately in the destination directories of all nodes where the application runs.

Data type	Boolean
Default	true

Classloader Mode

Specifies whether the class loader searches in the parent class loader or in the application class loader first to load a class. The standard for JDK class loaders and WebSphere Application Server class loaders is PARENT_FIRST. By specifying PARENT_LAST, your application can override classes contained in the parent class loader, but this action can potentially result in ClassCastException or LinkageErrors if you have mixed use of overridden classes and non-overridden classes.
The options are PARENT_FIRST and PARENT_LAST. The default is to search in the parent class loader before searching in the application class loader to load a class.

Data type	String
Default	PARENT_FIRST

WAR Classloader Policy

Specifies whether to use a single class loader to load all WAR files of this application or to use a different class loader for each WAR file.
The options are APPLICATION and MODULE. The default is to use a separate class loader to load each WAR file.

Data type	String
Default	MODULE

Create MBeans for Resources

Specifies whether to create MBean files for various resources (such as servlets or JSP files) within an application.

Data type	Boolean
Default	true

Reload Enabled

Specifies whether to enable class reloading when application files are updated.

Data type	Boolean
Default	false

Reload Interval

Specifies the number of seconds to scan the application's file system for updated files. The default is the value of the reload interval attribute in the IBM extension (`META-INF/ibm-application-ext.xmi`) file of the EAR file. This setting takes effect only if class reloading is enabled.
The reload interval specified here overrides the value specified in the IBM extensions for each Web module in the EAR file (which in turn overrides the reload interval specified in the IBM extensions for the application in the EAR file).

The range is from 0 to 2147483647.

Data type	Integer
Units	Seconds

Target mapping collection

Use this page to manage mappings of deployed applications or modules to servers or clusters.

To view this administrative console page, click **Applications > Enterprise Applications >***application_name* **> Target Mappings**.

Target:

States the name of the target server or cluster to which the application or module maps. You specify the target on the Map modules to application servers page accessed from the settings for an application.

Node:

Specifies the node name if the target is a server.

Status:

Indicates whether the status of the application running on the target server or cluster is started, stopped or unavailable.

Target mapping settings:

Use this page to map a deployed application or module to a server or cluster.

To view this administrative console page, click **Applications > Enterprise Applications >***application_name* **> Target Mappings >***target_name*.

Target:

States the name of the target server or cluster to which the application or module maps. You specify the target on the Map modules to application servers page accessed from the settings for an application.

Data type	String

Enabled:

Indicates whether the application modules installed on the target server are started (or enabled) when the server starts. This sets the initial state of application modules. A true value indicates that the corresponding modules are enabled and thus are accessible when the server starts. A false value indicates that the corresponding modules are not enabled and thus are not accessible when the server starts.

Data type	Boolean
Default	true

Starting and stopping applications

You can start an application that is not running (has a status of *Stopped*) or stop an application that is running (has a status of *Started*).

1. Go to the Enterprise Applications page. Click **Applications > Enterprise Applications** in the console navigation tree.
2. Check the check box for the application you want started or stopped.

3. Click a button:

Option	Description
Start	Runs the application and changes the state of the application from *Stopped* to *Started*.
Stop	Stops the processing of the application and changes the state of the application from *Started* to *Stopped*.

To restart a running application, place a check mark in the check box for the application you want to restart, click **Stop** and then click **Start**.

The status of the application changes and a message stating that the application started or stopped displays at the top the page.

Exporting applications

You can export an enterprise application to a location of your choice. Exporting applications enables you to back up your applications and preserve binding information for the applications. You might export your applications before updating installed applications or migrating to a later version of the WebSphere Application Server product.

1. Click **Applications > Enterprise Applications** in the administrative console navigation tree to access the Enterprise Applications page.
2. Place a check mark in the check box beside the application and click **Export**.
3. On the Export Application EAR Files page, click on the link to download the exported EAR file.
4. Use the browser dialogue to specify a location at which to save the exported EAR file and click **OK**.

The file containing binding information is exported to the specified node and directory, and has the name *enterprise_application_name*.ear.

Exporting DDL files

You can export the DDL files (Table.ddl) in the EJB modules of the application to a location of your choice.

1. Click **Applications > Enterprise Applications** in the administrative console navigation tree to access the Enterprise Applications page.
2. Place a checkmark in the check box beside the application and click **Export DDL**. If the application has no DDL files in any of its EJB modules, then the message *No DDL files were found* displays at the top of the page. If the application has DDL files in its EJB modules, then a page displays listing DDL files in the format appname.ear/_module.jar_Table.ddl.
3. Click on a file in the list to download the file to your machine.

Updating applications

You can update an application deployed on a server. The steps below describe how to update a deployed application using the administrative console.

Note: You can also update applications using the wsadmin tool, which provides updating capabilities identical to those available using the administrative console. Further, in some situations, you can update applications without needing to restart the application server.

1. Update the contents of the application and reassemble it, using the Application Assembly ToolApplication Assembly Tool. Typical tasks include adding or editing assembly properties, adding or importing modules into an application, and adding enterprise beans, Web components, and files.
2. Go to the Applications page of the administrative console. Click **Applications > Enterprise Applications** in the console navigation tree.
3. Back up the application. Place a checkmark in the check box beside the application you want uninstalled and click **Export** to export the application to an EAR file and preserve the binding information.
4. With a checkmark beside the application, click **Update**. The binding information of the updated (new) version of the application merges with the binding information from the installed (old) version. Then, the older version uninstalls from the configuration and the new version installs.
5. Complete the steps in the Preparing for application update page and the pages that follow it. See information on installing applications and on the settings page for application installation for guidance. Note that the installation steps have the merged binding information from the new version and the old version. If the new version has bindings for application artifacts such as EJB JNDI names, EJB references or resource references, then those bindings will be part of the merged binding information. If new bindings are not present, then bindings are taken from the installed (old) version. If bindings are not present in the old version and if the default binding generation option is enabled, then the default bindings will be part of the merged binding information. You can select whether to ignore bindings in the old version or ones in the new version.
6. Map the installed application or module to servers or clusters. Use the Map modules to application servers page of the Install New Application pages displayed during updating the application. Or, after updating the application, use the Map modules to application servers page accessed from the Enterprise Applications page.
   a. Go to the Map modules to application servers page. Click **Applications > Enterprise Applications** in the console navigation tree, click the application name, and then click **Map modules to application servers**.
   b. Specify the application server where you want to install modules contained in your application and click **OK**.
7. Click **Save** on the administrative console taskbar to save the changes to your configuration. In the single server (base) product, after you click **Save** the old version of the application is uninstalled and the new version is installed into the configuration. The application binaries for the old version are deleted from the destination directory and the new binaries are copied to the directory. In the Network Deployment product, the old application files are deleted and new files are copied when the configuration on the deployment manager synchronizes with the configuration on the node where the application is installed. If the application is running when you update it, the application stops running before its files are copied to the destination directory of the node and restarts after the copy operation completes. Thus, the application is unavailable on the node during the time the node is synchronizing its configuration with the deployment manager.

8. Restart the application so the changes take effect. If the application is updated while it is running, WebSphere Application Server stops the application, updates the application logic and restarts the application.
   a. Click **Applications > Enterprise Applications** in the console navigation tree to go to the Enterprise Applications page.
   b. Check the check box for the updated application.
   c. Click **Start**.
9. Optional: If the application you are updating is deployed on a server that has its application class loader policy set to SINGLE, restart the server.

Hot deployment and dynamic reloading

You can make various changes to applications and their contents without having to stop the server and start it again. Making these types of changes is known as *hot deployment and dynamic reloading*.

Hot deployment is the process of adding new components (such as WAR files, EJB Jar files, enterprise Java beans, servlets, and JSP files) to a running server without having to stop the application server process and start it again.

Dynamic reloading is the ability to change an existing component without needing to restart the server in order for the change to take effect. Dynamic reloading involves:
- Changes to the implementation of a component of an application, such as changing the implementation of a servlet
- Changes to the settings of the application, such as changing the deployment descriptor for a Web module

If the application you are updating is deployed on a server that has its application class loader policy set to SINGLE, you might not be able to dynamically reload your application. At minimum, you must restart the server after updating your application.

1. Locate your expanded application files. The application files are in the directory you specified when installing the application or, if you did not specify a custom target directory, are in the default target directory, *install_root*/installedApps/*cell_name*. Your EAR file, ${APP_INSTALL_ROOT}/*cell_name*/*application_name*.ear, points to the target directory. The variables.xml file for the node defines ${APP_INSTALL_ROOT}.

   It is important to locate the expanded application files because, as part of installing applications, a WebSphere application server unjars portions of the EAR file onto the file system of the computer that will run the application. These expanded files are what the server looks at when running your application.

   If you cannot locate the expanded application files, look at the binariesURL attribute in the deployment.xml file for your application. The attribute designates the location the run time uses to find the application files.

   For the remainder of this information on hot deployment and dynamic reloading, *application_root* represents the root directory of the expanded application files.
2. Locate application metadata files. The metadata files include the deployment descriptors (web.xml, application.xml, ejb-jar.xml, and the like), the bindings files (ibm-web-bnd.xmi, ibm-app-bnd.xmi, and the like), and the extensions files (ibm-web-ext.xmi, ibm-app-ext.xmi, and the like).

Metadata XML files for an application can be loaded from one of two locations. The metadata files can be loaded from the same location as the application binary files (such as *application_root*/META-INF) or they can be loaded from the WebSphere configuration tree, ${CONFIG_ROOT}/cells/*cell_name*/applications/*application_EAR_name*/deployments/*application_name*/. The value of the useMetadataFromBinary flag specified during application installation controls which location is used. If specified, the metadata files are loaded from the same location as the application binary files. If not specified, the metadata files are loaded from the application deployment folder in the configuration tree.

For the remainder of this information, *metadata_root* represents the location of the metadata files for the specified application or module.

3. CAUTION: If you are running WebSphere Application Server on a group of machines using Network Deployment and you are changing an application on a particular node, disable automatic synchronization.
   a. Click **System Administration > Node Agents** in the administrative console navigation tree, click on a node agent name, and then click **File Synchronization Service**.
   b. On the File Synchronization Service page, remove the checkmark from the check box for **Automatic Synchronization** and click **OK**.

   When you run WebSphere Application Server on a group of machines using Network Deployment and you change a file on the disk in the expanded application directory for a particular node, you can lose those changes the next time node synchronization occurs. In the Network Deployment environment, the configuration stored by the deployment manager is the master copy and any changes detected between that master copy and the copy on a particular machine trigger the master copy to be downloaded to the node.
4. Change or add the following components or modules as needed:
   - Application files
   - WAR files
   - EJB Jar files
   - HTTP plug-in configuration files
5. For changes to take effect, you might need to start, stop, or restart an application. ″“Starting and stopping applications” on page 911″ provides information on using the administrative console to start, stop, or restart an application. ″Example: Starting an application using wsadmin″ and ″Example: Stopping running applications on a server using wsadmin″ provide information on using the wsadmin scripting tool.
6. If you disabled automatic synchronization in step 3, return to the File Synchronization Service page, enable **Automatic Synchronization**, and click **OK**.

Changing or adding application files

You can change or add application files on application servers without having to stop the server and start it again. This file describes--
- Updating an existing application on a running server (providing a new EAR file)
- Adding a new application to a running server
- Removing an existing application from a running server
- Adding a new EJB or Web module to an existing, running application
- Changing the application.xml file for an application
- Changing the ibm-app-ext.xmi file for an application
- Changing the ibm-app-bnd.xmi file for an application
- Changing a non-module Jar file contained in the EAR file

- Update an existing application on a running server (providing a new EAR file). Reinstall an updated application using the administrative console or the wsadmin `$AdminApp install` command with the `-update` option

  Both reinstallation methods enable you to update an existing application using any of the other steps listed in this file, including changing classes, adding modules, removing modules, changing modules, or changing metadata files. The application reinstallation methods detect the changes in your application and prompt you for additional binding data that might be needed to install the application. The reinstallation process automatically stops and restarts your application on the appropriate servers.

Hot deployment: Yes
Dynamic reloading: Yes

- Add a new application to a running server. Install an application using the administrative console or the wsadmin `install` command.

Hot deployment: Yes
Dynamic reloading: No

- Remove an existing application from a running server. Stop the application and then uninstall it from the server. Use the administrative console to stop the application and then uninstall it. Or run the wasadmin `stopApplication` command and then the `uninstall` command.

Hot deployment: Yes
Dynamic reloading: No

- Add a new EJB or Web module to an existing, running application.
  1. Update the application files in the *application_root* location.
  2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Yes
Dynamic reloading: No

- Change the application.xml file for an application. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Change the ibm-app-ext.xmi file for an application. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Change the ibm-app-bnd.xmi file for an application. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and

`startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Change a non-module Jar file contained in the EAR file.
  1. Update the non-module Jar file in the *application_root* location.
  2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment: Yes
Dynamic reloading: Yes

Changing or adding WAR files

You can change WAR files on application servers without having to stop the server and start it again. This file describes--
- Changing an existing JSP file
- Adding a new JSP file to an existing application
- Changing an existing servlet class (editing and recompiling)
- Changing a dependent class of an existing servlet class
- Adding a new servlet using the Invoker (Serve Servlets by class name) facility or adding a dependent class to an existing application
- Adding a new servlet, including a new definition of the servlet in the web.xml deployment descriptor for the application
- Changing the web.xml file of a WAR file
- Changing the ibm-web-ext.xmi file of a WAR file
- Changing the ibm-web-bnd.xmi file of a WAR file

- Change an existing JSP file. Place the changed JSP file directly in the *application_root*/*module_name* directory or the appropriate subdirectory. The change will be automatically detected and the JSP will be recompiled and reloaded.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Add a new JSP file to an existing application. Place the new JSP file directly in the *application_root*/*module_name* directory or the appropriate subdirectory. The new file will be automatically detected and compiled on the first request to the page.

Hot deployment: Yes
Dynamic reloading: Yes

- Change an existing servlet class (editing and recompiling).
  1. Place the new version of the servlet .class file directly in the *application_root*/*module_name*/WEB-INF/classes directory. If the .class file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/lib. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class.

2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Change a dependent class of an existing servlet class.
  1. Place the new version of the dependent .class file directly in the *application_root*/*module_name*/WEB-INF/classes directory. If the .class file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/lib. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class.
  2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Add a new servlet using the Invoker (Serve Servlets by class name) facility or adding a dependent class to an existing application.
  1. Place the new .class file directly in the *application_root*/*module_name*/WEB-INF/classes directory. If the .class file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/lib. In either case, the change will be detected, the Web application will be shut down and reinitialized, picking up the new class. This case is treated the same as changing an existing class. The difference is that adding the servlet or class does not immediately cause the Web application to reload because the class has never been loaded before. The class simply becomes available for execution.
  2. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change.

Hot deployment: Yes
Dynamic reloading: Not applicable

- Add a new servlet, including a new definition of the servlet in the web.xml deployment descriptor for the application. Place the new .class file directly in the *application_root*/*module_name*/WEB-INF/classes directory. If the .class file is part of a Jar file, you can place the new version of the Jar file directly in *application_root*/*module_name*/WEB-INF/lib.

  You can edit the web.xml file in place or copy it into the *application_root*/*module_name*/WEB-INF/classes directory. The new .class file will not trigger a reloading of the application.
- Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands. After

the application restarts, the new servlet is available for service.

Hot deployment:	Yes
Dynamic reloading:	Not applicable

- Change the web.xml file of a WAR file.
  1. Edit the web.xml file in place or copy it into the *metadata_root*/*module_name*/WEB-INF directory.
  2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Yes
Dynamic reloading:	Yes

- Change the ibm-web-ext.xmi file of a WAR file. Edit the extension settings as needed. You can change all of the extension settings. The only warning is if you set the reloadInterval property to zero (0) or the reloadEnabled property to false, the application will no longer automatically detect changes to class files. Both of these changes disable the automatic reloading function. The only way to re-enable automatic reloading is to change the appropriate property and restart the application. See other task descriptions in this file for information on restarting an application.

Hot deployment:	Not applicable
Dynamic reloading:	Yes

- Change the ibm-web-bnd.xmi file of a WAR file.
  1. Edit the bindings as needed. You can change all of the values but ensure that the entities you are binding to are present in the configuration of the server.
  2. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Not applicable
Dynamic reloading:	Yes

Changing or adding EJB Jar files

You can change EJB Jar files on application servers without having to stop the server and start it again. This file describes--

- Changing the ejb-jar.xml file of an EJB Jar file
- Changing the ibm-ejb-jar-ext.xmi or ibm-ejb-jar-bnd.xmi file of an EJB Jar file
- Changing the Table.ddl file for an EJB Jar file
- Changing the Map.mapxmi or Schema.dbxmi file for an EJB Jar file
- Updating the implementation class for an EJB file or a dependent class of the implementation class for an EJB file
- Updating the Home/Remote interface class for an EJB file
- Adding a new EJB file to an existing EJB Jar file

- Change the ejb-jar.xml file of an EJB Jar file. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Not applicable

Dynamic reloading: Yes

- Change the ibm-ejb-jar-ext.xmi or ibm-ejb-jar-bnd.xmi file of an EJB Jar file. Restart the application. Automatic reloading will not detect the change. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Change the Table.ddl file for an EJB Jar file. Rerun the DDL file on the user database server. Changing the Table.ddl file has no effect on the application server and is a change to the database table schema for the EJB files.

Hot deployment: Not applicable
Dynamic reloading: Not applicable

- Change the Map.mapxmi or Schema.dbxmi file for an EJB Jar file.
  1. Change the Map.mapxmi or Schema.dbxmi file for an EJB Jar file.
  2. Regenerate the deployed code artifacts for the EJB file.
  3. Apply the new EJB Jar file to the server.
  4. Restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Update the implementation class for an EJB file or a dependent class of the implementation class for an EJB file.
  1. Update the class file in the *application_root*/*module_name*.jar file.
  2. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change. If automatic reloading is not enabled, restart the application of which the EJB file is a member. If the updated module is used by other modules in other applications, restart those applications as well. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Update the Home/Remote interface class for an EJB file.
  1. Update the interface class of the EJB file.
  2. Regenerate the deployed code artifacts for the EJB file.
  3. Apply the new EJB Jar file to the server.
  4. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change. If automatic reloading is not enabled, restart the application of which the EJB file is a member. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment: Not applicable
Dynamic reloading: Yes

- Add a new EJB file to an existing EJB Jar file.
  1. Apply the new or updated Jar file to the *application_root* location.
  2. If automatic reloading is enabled, you do not need to take further action. Automatic reloading will detect the change. If automatic reloading is not enabled, restart the application. Use the administrative console to restart the application. Or run the wasadmin `stopApplication` and `startApplication` commands.

Hot deployment:	Yes
Dynamic reloading:	Yes

Changing the HTTP plug-in configuration

You can change the HTTP plug-in configuration without having to stop the server and start it again. This file describes--
- Changing the application.xml file to change the context root of a WAR file
- Changing the web.xml file to add, remove, or modify a servlet mapping
- Changing the server.xml file to add, remove, or modify an HTTP transport or changing the virtualhost.xml file to add or remove a virtual host or to add, remove, or modify a virtual host alias

Changing the application.xml file to change the context root of a WAR file
1. Change the application.xml file.
2. Regenerate the plug-in configuration file using the administrative console or by running the GenPluginCfg.bat/sh script.

Hot deployment:	Yes
Dynamic reloading:	No

Changing the web.xml file to add, remove, or modify a servlet mapping
1. Change the web.xml file.
2. Regenerate the plug-in configuration file using the administrative console or by running the GenPluginCfg.bat/sh script.

   If the Web application has file serving enabled or has a servlet mapping of `/`, you do not have to regenerate the plug-in configuration. In all other cases the regeneration is required.

Hot deployment:	Yes
Dynamic reloading:	Yes

Changing the server.xml file to add, remove, or modify an HTTP transport or changing the virtualhost.xml file to add or remove a virtual host or to add, remove, or modify a virtual host alias
1. Change the server.xml file to add, remove, or modify an HTTP transport or change the virtualhost.xml file to add or remove a virtual host or to add, remove, or modify a virtual host alias.
2. Regenerate the plug-in configuration file using the administrative console, by running the GenPluginCfg.bat/sh script, or by running a wsadmin command.

Hot deployment:	Yes
Dynamic reloading:	Yes

Uninstalling applications

After an application no longer is needed, you can uninstall it. Uninstalling an application deletes the application from the WebSphere Application Server configuration repository and it deletes the application binaries from the file system of all nodes where the application modules are installed.

1. Click **Applications > Enterprise Applications** in the administrative console navigation tree to access the Enterprise Applications page.
2. Stop the application. Select the application you want uninstalled and click **Stop**.
3. Back up the application. Select the application you want uninstalled and click **Export** to export the application to an EAR file and preserve the binding information.
4. Select the application you want uninstalled and click **Uninstall**.
5. Click **Save** on the console taskbar to save changes made to the administrative configuration.

In the single-server (base) product, application binaries are deleted after you click **Save**.In the Network Deployment product, application binaries are deleted when configuration changes on the deployment manager synchronize with configurations for individual nodes.

Deploying and managing applications: Resources for learning

Use the following links to find relevant supplemental information about deploying and managing applications using the administrative console. The information resides on IBM and non-IBM Internet sites, whose sponsors control the technical accuracy of the information.

These links are provided for convenience. Often, the information is not specific to the IBM WebSphere Application Server product, but is useful all or in part for understanding the product. When possible, links are provided to technical papers and Redbooks that supplement the broad coverage of the release documentation with in-depth examinations of particular product areas.

View links to additional information about:
- Programming model and decisions
- Programming instructions and examples
- Programming instructions and examples

Programming model and decisions
- The J2EE™ Tutorial: The Duke's Bank Application
- Best Practices in WebSphere Application: Separating the developers from the administrators
- Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, Second Edition
- Designing Enterprise Applications, Second Edition
- Building Java™ Enterprise Applications Volume I: Architecture

Programming instructions and examples
- WebSphere Application Server education
- Developing and Testing a Complete 'Hello World' J2EE Application with IBM WebSphere Studio Application Developer for Linux
- Writing Enterprise Applications with Java™ 2 Platform, Enterprise Edition

Administration
- Listing of all IBM WebSphere Application Server Redbooks

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

© Copyright IBM Corp. 2003

Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX
- CICS
- Cloudscape
- DB2
- DFSMS
- Everyplace
- iSeries
- IBM
- IMS
- Informix
- iSeries
- Language Environment
- MQSeries
- MVS
- OS/390
- RACF
- Redbooks
- RMF
- SecureWay
- SupportPac
- ViaVoice
- VisualAge
- VTAM
- WebSphere
- z/OS
- zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

© Copyright IBM Corp. 2003