WebSphere® IBM WebSphere Application Server Version 7.0

Feature Pack for Service Component Architecture (SCA) Version 1.0.0 information center topics

Note
Before using this information, be sure to read the general information under Appendix A, "Notices," on page 193.

Compilation date: December 11, 2009

Contents

Chapter 1. Version 1.0.0 topics 1

Chapter 2. SCA in WebSphere Application Server: Overview 3
What is new in the Feature Pack for SCA 5
Learn about SCA composites 6
SCA components 7
SCA composites 8
SCA domain 10
SCA contributions 10
SCA Samples 13
Feature packs 14

Chapter 3. Specifications and API documentation 17

Chapter 4. Unsupported SCA specification sections 29

Chapter 5. Developing Service Component Architecture (SCA) services and applications 33
Developing SCA services from existing WSDL files 33
Developing SCA services with existing Java code . . 38
Developing SCA service clients 42
Using business exceptions with SCA interfaces . . 49
Considerations for developing SCA applications using EJB bindings 53

Chapter 6. Specifying bindings in an SCA environment 57
Configuring the SCA default binding 58
Using the SCA default binding to find and locate SCA services 61
Configuring the SCA Web service binding 61
Configuring Web service binding custom endpoints to support a proxy server 67
Routing requests to an SCA service exposed over the SCA Web service binding when using external Web servers 68
Using EJB bindings in SCA applications 70
Using EJB bindings in SCA applications in a cluster environment 73
Resolving SCA references 75

Chapter 7. Using JAXB for XML data binding 79
Using JAXB schemagen tooling to generate an XML schema file from a Java class 81
Using JAXB xjc tooling to generate JAXB classes from an XML schema file 85

Chapter 8. SCA application package deployment 89

Chapter 9. Creating SCA business-level applications 91
Creating SCA business-level applications with the console 92
Map virtual host settings for SCA composites . . 94
Attach policy set settings 95
Map security roles to users or groups collection for SCA composites 97
Map RunAs roles to users collection for SCA composites 99
Composition unit settings for SCA composites 100
Provide HTTP endpoint URL information settings for SCA composites 102
SCA composite component settings 102
SCA component reference settings 103
SCA component service settings 104
Service provider policy sets and bindings collection for SCA composites 105
References policy sets and bindings collection for SCA composites 107
SCA service provider settings 109
SCA service client settings 112
Example: Creating an SCA business-level application with the console 116

Chapter 10. Updating SCA composite artifacts 119

Chapter 11. Viewing SCA composite definitions 121

Chapter 12. Viewing SCA domain information 123

Chapter 13. Deleting business-level applications 125

Chapter 14. Administering applications using wsadmin scripting . 127
Setting up business-level applications using wsadmin scripting 127
Example: Creating an SCA business-level application with scripting 130
Deleting business-level applications using wsadmin scripting 133

Chapter 15. Managing deployed applications using wsadmin scripting . 135
Exporting SCA domain information using scripting 135

Exporting WSDL and XSD documents using scripting 137

Chapter 16. Authorizing access to resources 141
Using SCA authorization and security identity policies 141
Using the SCA RequestContext.getSecuritySubject() API. 143

Chapter 17. Using JAXB for XML data binding 147
Using JAXB schemagen tooling to generate an XML schema file from a Java class 149
Using JAXB xjc tooling to generate JAXB classes from an XML schema file 153

Chapter 18. Defining and managing secure policy set bindings. 157
Configuring Web service binding for SCA transport layer authentication 157
Configuring Web service binding to use SSL . . . 158
Configuring Web service binding for LTPA authentication 159

Chapter 19. Mapping abstract intents and managing policy sets 161
Attached deployed assets collection 164
Name 164
Type 165

Chapter 20. Administering asynchronous beans 167
Configuring work managers 167
Configuring Work managers for one-way operations 169
Configuring the default SCA Work manager for the SCA layer 171

Chapter 21. Transaction support in WebSphere Application Server 173
SCA transaction intents 175

Chapter 22. Dynamic cache service eviction policies 179
Eviction policies using the disk cache garbage collector 179
Example: Caching Web services 180

Chapter 23. Using PassByReference optimization in SCA applications . . . 185

Chapter 24. Directory conventions 187

Appendix A. Notices 193

Appendix B. Trademarks and service marks. 195

Chapter 1. Version 1.0.0 topics

This book contains information center topics that describe how to use Feature Pack for SCA Version 1.0.0, which is a feature pack of the IBM® WebSphere® Application Server Version 7.0 product.

The topics describe how to use the feature pack functionality after installation of the feature pack. For information about installing Feature Pack for SCA Version 1.0.0 and creating a feature pack profile, see "Getting started with the Feature Pack for Service Component Architecture (SCA) Version 1.0" (GettingStarted_SCA_V10.pdf).

Feature Pack for SCA Version 1.0.0 is supported on AIX®, HP-UX, IBM i, Linux®, Solaris, Windows®, and z/OS® operating systems. Users of this product on an IBM i operating system must use Version 1.0.0. Feature Pack for SCA Version 1.0.1 is not supported on an IBM i operating system.

Chapter 2. SCA in WebSphere Application Server: Overview

IBM WebSphere Application Server V7 Feature Pack for SCA delivers critical technology that enables adoption of key Service-Oriented Architecture (SOA) principles.

As part of the larger SOA Foundation, which straddles all of IBM software brands, this offering delivers an integrated, open implementation of Service Component Architecture (SCA), a technology specified by IBM and other industry leaders through the Open SOA Collaboration (OSOA).

WebSphere has taken the open source implementation from Tuscany, an Apache project, and integrated it with WebSphere Network Deployment. This integration ensures that all of WebSphere's capabilities work together with SCA applications to provide a compatible environment for both the SCA and existing applications.

IBM WebSphere Application Server V7 Feature Pack for SCA, as well as the underlying Tuscany framework, is a ″proof-point″ delivery of SCA built using a plug-in concept, allowing for additional plug-in capabilities in subsequent releases.

The primary objective of this release is to highlight usage of SCA as a coarse-grained composition model that can be used to assemble and compose existing services in your enterprise. The key principle of SOA demonstrated by this support is the ability to use your existing services to create new ones.

Another key objective of this delivery of SCA is to highlight the ease-of-use characteristics of SCA service development in Java™. This is accomplished by demonstrating annotated Plain-Old Java-Object (POJO) components deployed using simple JAR packaging schemes, an easy to use assembly model, and powerful wiring abstractions that enable service definition over different transports and protocols. The key principle of SOA demonstrated by this support is having the right information to get the job done.

Service composition

Businesses today are challenged not only by competitors, but by social and economic pressures that directly affect their information technology systems. As businesses adopt SOA and build a growing inventory of business services, there is a real need to be able to compose, reuse, and otherwise assemble new services from those existing business services.

SCA offers a metadata assembly and composition model for assembling and constructing coarse-grained services out of software components and other services. In a sense, SCA applies the hardware circuit-board paradigm to software programming, in that while service implementations are vital to the functioning of the overall application, they can also be viewed as “chips” because details of their inner workings are hidden from the assembler.

SCA assemblies provide the metadata language to describe the chips, hide certain details, and provide wiring and binding semantics to the workings inside the chips, as well as to those exposed outside the chip. In the hardware realm, wires have physical constraints, such a certain voltage range or frequency of operation. Software “wires” have similar constraints which are expressed as SCA policy.

Service development

SCA has a language-neutral programming model for which there are multiple language-specific specifications defined at OSOA. The language-specific component models include Java, Spring, and C++. Being a Java runtime environment, WebSphere Application Server supports SCA in Java in a highly natural way.

The concepts of SCA apply broadly across both Java and non-Java application environments. The SCA component model has at its heart a strong focus on a proper separation of concerns. The service consumer business logic author does not need to know the details of the service implementation. For instance, a Java service consumer does not have to know that a target service is implemented using C++ or COBOL.

A logical name can be used to identify a service, for example, *MyStockQuoteService*, and that name can be used in SCA wires to delegate the specific details of service connectivity to the SCA runtime environment. Essentially, the application programmer is telling SCA to use the best connectivity alternatives available to wire the service consumer to the service providing *MyStockQuoteService*.

Service agility and flexibility

One of the key reasons for SOA is to provide a set of patterns and best practices–formalized through infrastructural concepts and products--that allows businesses to realign and remission multiple aspects of information technology. IT professionals expect to be able to rewire, recast, and reuse applications in flexible ways to keep up with business needs in the dynamic business climates of today.

This release of Feature Pack for SCA highlights the flexibility and agility of metadata bindings, and the appropriate separation of concerns. The ability to rewire, compose, and assemble business logic without impacting the business logic itself is key.

OSOA support

As mentioned previously, this delivery of SCA in WebSphere Application Server follows the definition of the technology as documented by OSOA. Defining a set of compliance test suites was not part of the OSOA charter, so the implementation provided in this feature pack uses the following specifications as guiding principles. However, IBM provides an implementation that adheres strictly to our interpretation of the specifications listed below:

- SCA Assembly Model
- SCA Java Component Implementation
- SCA Java Annotations and API
- SCA Transaction Policy
- SCA Web Service Binding
- SCA EJB Binding
- SCA Policy Framework

See the ″Unsupported SCA specifications sections″ topic for restrictions and limitations that are unsupported at this time.

WebSphere support for SCA

As already noted, multiple specifications are defined at OSOA, as well as Tuscany extensions provided in open source that go beyond the basic mission of WebSphere Application Server. Each vendor can decide which aspects of SCA apply to their product. For WebSphere Application Server, the focus is on enabling compositions as services, Java components, and integration of key qualities of service-like transactions and security.

SCA can enable mediations, business rules, and business process execution language to be treated as any other service, and while WebSphere Application Server provides the mechanisms to wire services that are implemented in those languages and environments, the product does not provide native support to host those kinds of service implementations.

What is new in the Feature Pack for SCA

The Feature Pack for Service Component Architecture (SCA) Version 1.0 is an optionally installable product extension for IBM WebSphere Application Server Version 7.*x* that offers a simple and powerful way to construct applications based on Service-Oriented Architecture (SOA). This feature pack leverages the Apache Tuscany open-source technology to provide an implementation of the published SCA specifications.

Benefits of the Feature Pack for SCA

Through this feature pack, your organization will be able to move quickly into the world of SOA, as follows:

Improve flexibility in application deployment

- Adapt applications quickly to reflect changes in the business environment
- Reuse the components you create in other business processes and composite applications
- Easily compose services into more complex composite applications
- Adjust solutions to accommodate varying technology offerings (that is, protocols or deployment targets) without the need to rebuild business applications

Increase programmer productivity

- Stay focused on solving business problems, rather than getting bogged down in the individual complexities of the technologies that connect service consumers and service providers
- Use the same fundamental principles to uniformly represent existing assets and newly engineered components
- Organize service components into logical modules to hasten composite application development
- Leverage the loosely coupled service model with clear service definitions to enable developers to work independently and in parallel, for fast delivery of solutions

SCA support

SCA support includes the following:

- POJO (Java Object) service-component implementations, including support for annotations
- Asynchronous capability
- Recursive composition model support
- Several binding types, including Web services binding, SCA default binding, and Enterprise JavaBeans™ (EJB) binding
- Preview of native SCA deployment
- Sample SCA composites compiled specifically for use with this feature pack

New and changed features in the Feature Pack for SCA

Notable changes to WebSphere Application Server Version 7.*x* provided by the Feature Pack for SCA include the following:

- Support for SCA specifications
- Support for SCA services developed from existing WSDL files or Java code
- Support for SCA, Web service, and EJB bindings
- Support for Java Architecture for XML Binding (JAXB) data bindings in SCA applications
- Deployment of SCA composites in business-level applications
- SCA authorization and security identity policies
- PassByReference optimization for SCA applications

:

Learn about SCA composites

Find links to Service Component Architecture (SCA) resources for learning, including conceptual overviews, tutorials, samples, and ″How do I?...″ topics, pending their availability.

How do I?...

Develop SCA composites

- Develop SCA service clients
- Develop SCA services with existing Java code
- Develop SCA services from existing WSDL files
- Specify bindings for SCA components

Deploy SCA composites in business-level applications

- Create SCA business-level applications using the administrative console
- Create SCA business-level applications using wsadmin scripts

Administer deployed SCA composites

- Update deployment settings for SCA composites
- Export SCA domain information
- Export WSDL and XSD documents
- Delete business-level applications

Conceptual overviews

- SCA overview
- SCA components
- SCA composites
- Support for SCA composite deployment

Samples

Feature Pack for SCA offers sample files that support SCA specifications. You can use these sample SCA files in business-level applications. The sample files are located in *app_server_root*/samples/SCA and *app_server_root*/installableApps. SCA services are packaged in Java archive (JAR) files that you import as assets to the product repository and then add as composition units to business-level applications.

Each sample includes detailed deployment instructions in a readme.html file in a *app_server_root*/samples/SCA subdirectory. See "SCA Samples."

SCA components

A Service Component Architecture (SCA) component is a configured instance of an implementation, which is program code that implements one or more business functions such as Java classes. Components provide and consume services. The business functions provide services. Components consume services by referring to services provided by other components. The component configuration sets values for properties that are declared by the implementation and specifies references that point to services provided by other components.

The SCA component graphic shows the parts of a component:

- The green chevron pointing towards the component represents a service, or business function, that the component provides to its client.
- The purple chevron pointing away from the component represents a reference to a service provided by another component.
- The yellow box on the component represents a property value for a property that is declared by the implementation. The component reads the property value from the configuration file when the component is instantiated.

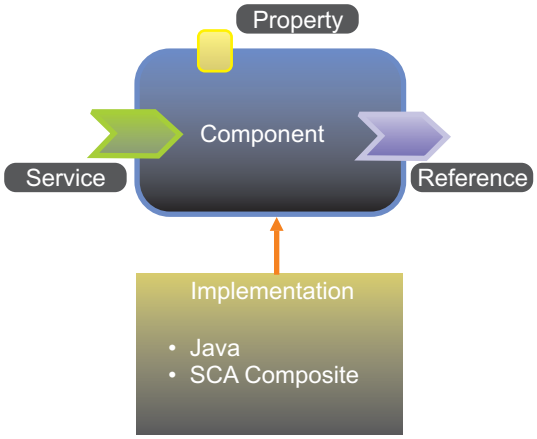


Figure 1. SCA component

The implementation defines the service in code that is appropriate for the chosen language. For example, a Java component might describe its service using Java interfaces. Supported implementations include Java Pojo and SCA composites.

More than one component can use the same implementation.

SCA composites

A Service Component Architecture (SCA) composite is a composition unit within an SCA domain. An SCA composite can consist of components, services, references, and wires that connect them. A composite is the unit of deployment for SCA.

The SCA composite graphic shows the parts of a composite and its components:

- The blue boxes on the composite represent components. A composite can have one or multiple components.
- A green chevron pointing towards a component represents a service, or business function, that a component provides to its client.
- A purple chevron pointing away from a component represents a reference to a service provided by another component.
- The yellow box on a component represents a property value for a property that is declared by a component implementation.
- The white solid line from the reference of one component to the service of another component represents a wire. A wire between a Component1 reference and a Component2 service indicates that Component1 requires the service provided by Component2.
- An implementation defines a component service in code that is appropriate for the chosen language. Supported implementations include Java Pojo and SCA composites.

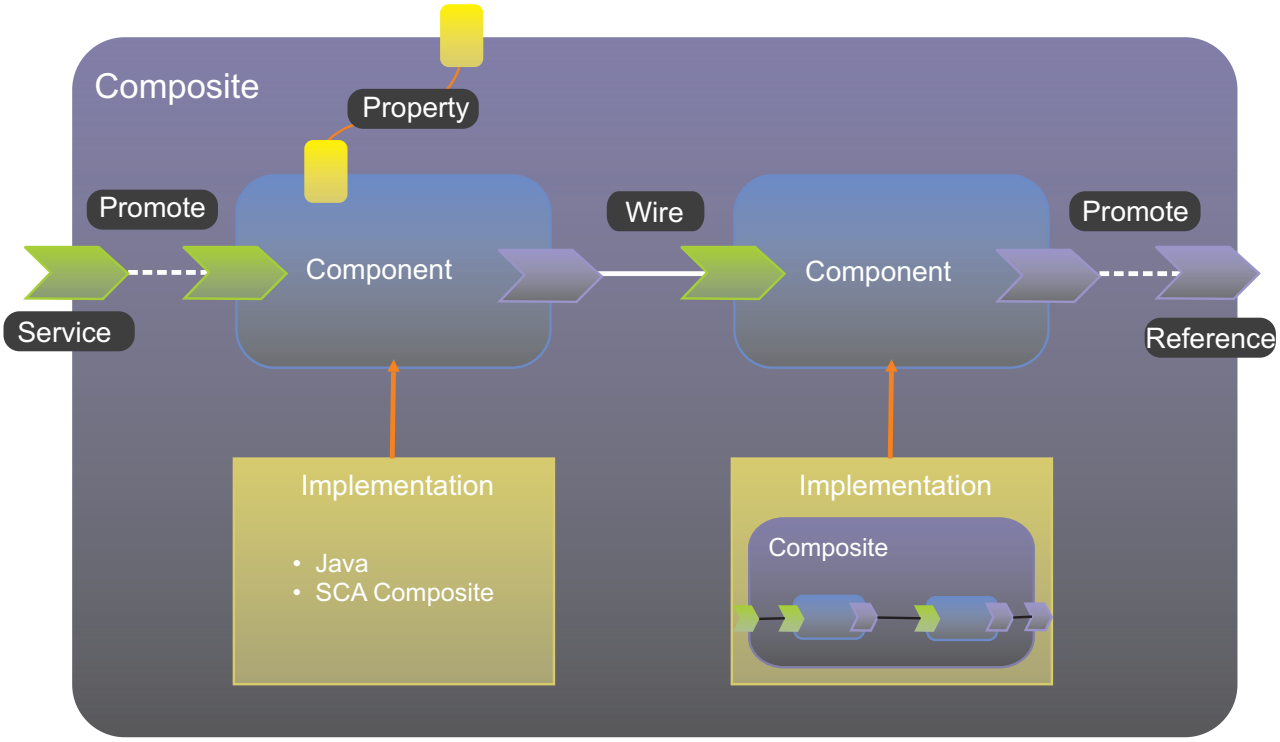


Figure 2. SCA composite

An application can contain one composite or several different composites. The components of a composite can run in a single process on a single computer or be distributed across multiple processes on multiple computers. The components might all use the same implementation language, or use different languages.

An SCA composite is typically described in a configuration file, the name of which ends in `.composite`. The SCA composite parts mapped to the helloworldws composite file graphic shows the composite definition of the helloworldws composition unit in the SCA sample application HelloWorldAsync. You can find the composite definition for the helloworldws composition unit in the `/META-INF/sca-deployables/default.composite` file.

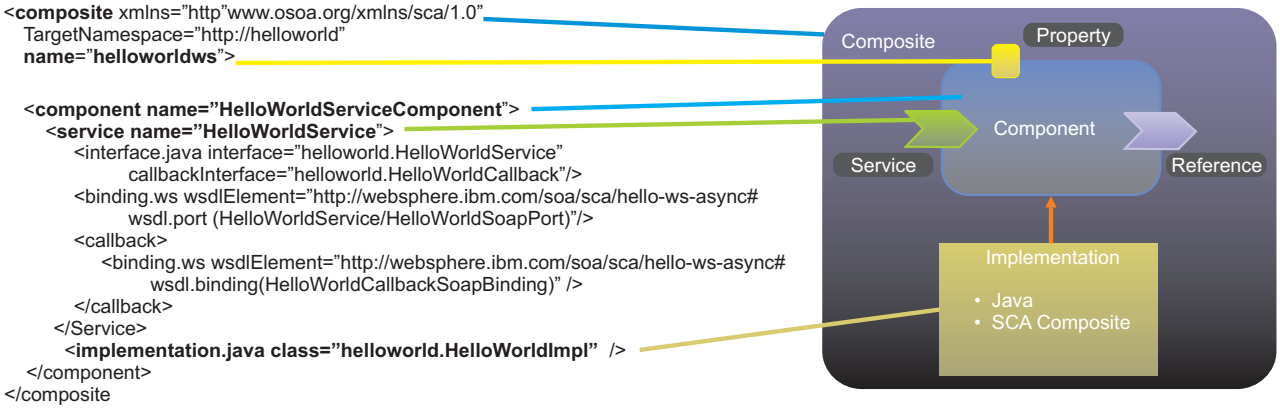


Figure 3. SCA composite parts mapped to the helloworldws composite file

In the Feature Pack for SCA, a composite file in a WAR file must be named `default.composite`. A composite file that is not in a WAR file can have any name.

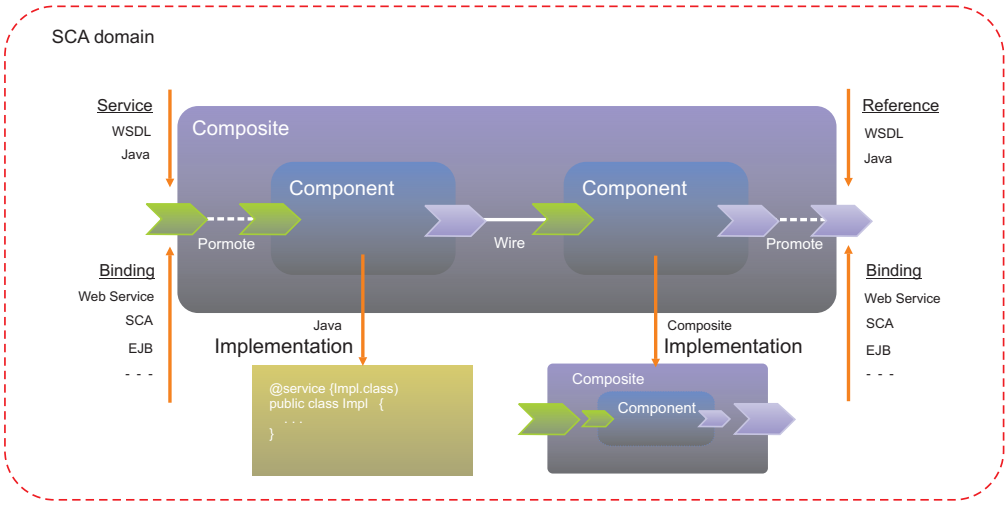
The Feature Pack for SCA supports composite as an implementation, as described in Section 1.6 of SCA Assembly Specification 1.0. See "Unsupported SCA specification sections" for information on parts of Section 1.6 that the feature pack does not support.

SCA domain

A Service Component Architecture (SCA) domain consists of the definitions of composites, components, their implementations, and the nodes on which they run. Components deployed into a domain can directly wire to other components within the same domain. For the Feature Pack for SCA on a single server, the domain is essentially the scope of the server. For a multiple-server configuration, the domain is essentially the cell.

The SCA domain in which composites reside graphic shows one composite in an SCA domain.

Figure 4. SCA domain in which composites reside

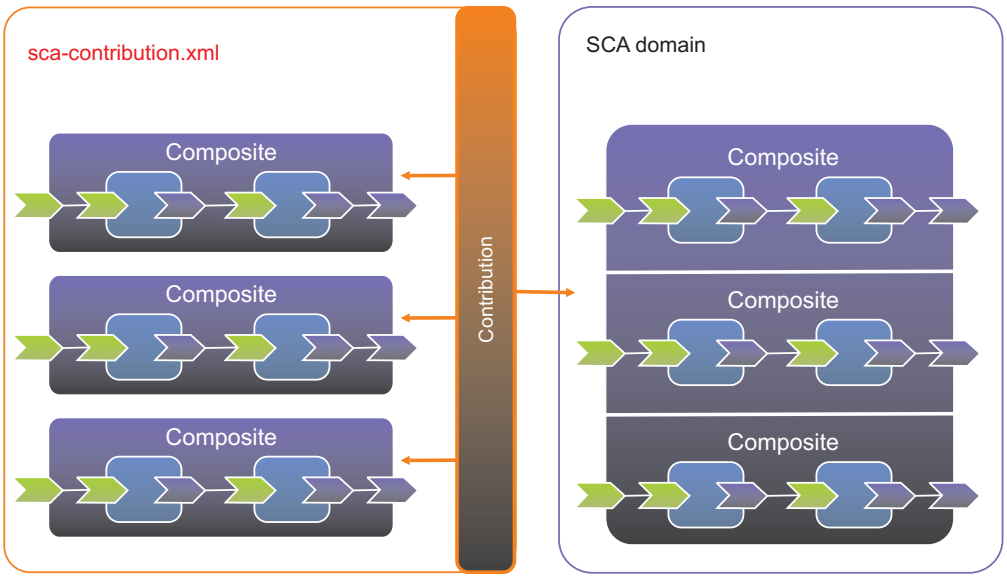


SCA contributions

A Service Component Architecture (SCA) contribution contains artifacts that are needed for an SCA domain. Contributions are sometimes self-contained, in that all of the artifacts necessary to run the contents of the contribution are found within the contribution itself. However, the contents of the contribution can make one or many references to artifacts that are not contained within the contribution. These references might be to SCA artifacts, or to other artifacts, such as Web Services Description Language (WSDL) files, XSD files, or to code artifacts such as Java class files.

The SCA contribution graphic shows composites in an `sca-contribution.xml` file in an SCA domain.

Figure 5. SCA contribution



An SCA contribution is typically described in a contribution file, named `sca-contribution.xml` in the `META-INF` directory. The contribution file for the helloworldws composition unit in the SCA sample application HelloWorldAsync follows:

```
<?xml version="1.0" encoding="ASCII"?>
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0" xmlns:ns="http://helloworld">
   <deployable composite="ns:helloworldws"/>
</contribution>
```

The Feature Pack for SCA supports contributions, as described in Chapter 1.10 of SCA Assembly Specification 1.0. The assembly specification defines the contribution metadata model to describe the runnable components for a given contribution, as well as the exported definitions and imported definitions to help resolve artifact dependencies across multiple contributions. See "Unsupported SCA specification sections" for information on parts of Chapter 1.10 that the feature pack does not support.

Information about support for SCA contributions follows:

- Preconditions or inputs
- Postconditions or outputs for JAR packaged applications
- Postconditions or outputs for WAR packaged applications
- Contribution scenarios
- Scenarios for mapping multiple deployable composites to a composition unit
- Notes and limitations

Preconditions or inputs

One SCA contribution, including multiple deployable composites, and with import or export definitions.

Postconditions or outputs for JAR packaged applications

- After successful creation of an SCA contribution as an asset, you can do the following:
  - Create a business-level application, add the asset as a composition unit, and start and stop the application.
  - Target each deployable composite to a different server or cluster.

  - Delete the deployable composites individually.
- Support import or export namespace:
  - WSDL can be defined in another contribution. Use `<import name = "name_ space"/>` to declare dependencies.
  - Schema XSD can be defined in another contribution. Use `<import name = "name_space"/>` to declare dependencies
  - Use `<export name="name_space"/>` to make WSDL or XSD available to other contributions.
  - Composite for recursive model can be defined in another contribution.
- Support import.java and export.java package name:
  - The Java package can be in another contribution. Use `<import.java name="java_ package_ name">` to declare dependencies.
  - Use `<export.java name="java_ packages">` to make Java packages available to other contributions.

Postconditions or outputs for WAR packaged applications

Successful installation of a WAR module with a single deployable composite in the contribution.

Contribution scenarios

- There are multiple runnable deployable composites in a contribution. One extreme case is that there is no runnable for the contribution, so the contribution is used as a shared library for resources and classes.
- Declare an import or export namespace for resource resolution, such as WSDL, XSD, and composite definition.
- Declare import.java and export.java for classLoader dependencies.
- Support artifact resolution for contributions targeted on the same server.
- Support artifact resolution for contributions targeted to a different server or cluster in a multiple-server environment.
- Support the partial update of an installed contribution (asset). This can be anything other than a change to `.composite` or `sca-contribution.xml`. You should be able to restart the composition unit that has the assets as a dependant.

Scenarios for mapping multiple deployable composites to a composition unit

- Scenario 1:
  - A contribution can contain multiple deployable composites.
  - During asset creation, each deployable composite is identified as a deployable unit.
  - When creating a composition unit, you can select only one deployable unit. For SCA composition units, you cannot select multiple deployable composites. This is different from non-SCA business-level applications, for which you can select multiple deployable units.
  - After composition unit creation, each composition unit is mapped to one composite.
- Scenario 2:

  A Scenario 1 variation consists of an asset with an `sca-contribution.xml` file that has zero-to-many (0...n) deployable composites:

– Use `sca-contribution.xml` deployable composites to create 0...n deployable units with the deployable composite's QName as the deployable unit name.
– The asset is tagged as an SCA asset.
– If the composition unit is created by selecting one, and only one, of the deployable units:
  - View or edit targets the deployable unit's composite.
  - Start or stop targets the deployable unit's composite.
– If the composition unit is created as a shared library, or no deployable unit is selected, the default deployable unit is used:
  - View or edit is not available for the composition unit.
  - Start or stop is not available for the composition unit.

Notes and limitations

Currently this topic focuses on JAR-packaged SCA applications. For WAR-packaged applications support is provided for only a single deployable composite in the contribution.

SCA Samples

The product offers sample files that support Service Component Architecture (SCA) specifications. You can use these sample SCA files in business-level applications. The sample files are located in *app_server_root*`/samples/SCA` and *app_server_root*`/installableApps`. SCA services are packaged in Java archive (JAR) files that you import as assets to the product repository and then add as composition units to business-level applications. Each sample includes detailed deployment instructions in a `readme.html` file in a *app_server_root*`/samples/SCA` subdirectory.

Samples installation

To use the Samples, install the Feature Pack for SCA. Installing the feature pack adds SCA sample files to the *app_server_root*`/installableApps` directory. If you selected to install Samples during creation of a profile enabled by the feature pack, the product also adds several SCA sample files to the *app_server_root*`/samples/SCA` directory. You must deploy SCA sample files as assets of a business-level application to a Version 7.0 server or cluster that is enabled for the Feature Pack for SCA.

Samples description

CandyStore

The `CandyStore.jar` file uses the default binding, Web service binding, and EJB binding and shows the use of the recursive model, and authentication and authorization over the default binding. The sample shows both the bottom-up (Java to WSDL) and top-down (WSDL to Java) approaches in developing SCA applications.

For a description of the 10 composites in CandyStore and details on deployment, refer to *app_server_root*`/samples/SCA/CandyStore/documentation/readme.html`.

HelloWorldAsync

The `helloworld-ws-asynch.jar`, `helloworld-ws-client-asynch.war`, and `helloworld-ws-asynchclient.jar` files use Web services. One client uses a Java ServerPages (JSP) file to obtain an SCA composite context and invoke the HelloWorldClient service over an SCA default binding. The client

service then invokes the HelloWorld service over a Web services binding. After the service is invoked, the service does a callback to the client service. The client JSP waits for 5 seconds for the callback to complete and then displays the callback result.

For a description of the JAR files and details on deployment, refer to *app_server_root*/samples/SCA/helloworld-ws-asynch/documentation/readme.html. For detailed instructions on deploying the helloworld-ws-asynch.jar file in a business-level application, refer to "Example: Creating an SCA business-level application with the console" and "Example: Creating an SCA business-level application with scripting."

JobbankTargetEJBApp

This sample shows how a Java EE client, JobbankClientApp.ear, can invoke an SCA component, jobbankejb.jar, using a remote EJB service binding as well as how a component with an EJB reference binding can invoke a remote external EJB, which is in JobbankTargetEJBApp.ear. The stateless session bean binding is a protocol binding that provides the ability to integrate SCA with EJB based services. The SCA support is in jobbankejb.jar.

For a description of the files and details on deployment, refer to *app_server_root*/samples/SCA/jobbankejb/documentation/readme.htm.

MultiService

This sample shows service composition using existing services. The MultiService sample wires to several existing samples:

- Stock Quote, WebServicesSamples.ear
- HelloWorldAsync, helloworld-ws-asynch.jar
- JobbankTargetEJBApp, JobbankTargetEJBApp.ear
- EJB Counter, EJB3CounterSample.ear

You can find the files in *app_server_root*/installableApps.

For details on deployment, refer to *app_server_root*/samples/SCA/MultiService/documentation/readme.html.

MyBank

This sample shows how to create an SCA application that uses JAXB following a top-down approach. You use AccountService.wsdl to generate JAXB classes that provide a data binding between XML and Java files. You then use a reference on the client side WAR file to wire to the Account Service over the Web service binding.

For details on deployment, refer to *app_server_root*/samples/SCA/MyBank/documentation/readme.html.

Feature packs

WebSphere Application Server feature packs are a mechanism for providing major new application server function between product releases. By delivering new functions and support for industry standards between product releases, you can more quickly explore and implement new technologies within your business applications in today's rapidly changing business environments.

WebSphere Application Server feature packs are optionally installable product extensions that offer targeted, incremental new features. WebSphere Application

Server customers who wish to take advantage of these features can download the appropriate feature pack and install it on one, or all, of their entitled application servers.

The primary characteristics of feature packs are:

- Feature packs provide production-ready function (such a new open standards) to customers who need them, without having to wait for a new WebSphere Application Server release. Some feature packs may be merged into later releases of the WebSphere Application Server product.
- In WebSphere Application Server Version 7.0, feature packs are free of charge to WebSphere Application Server Version 7.0 customers.
- Customers can choose which feature packs, if any, they wish to install. In some cases, one feature pack may rely on the capabilities of another feature pack, in which case both must be installed.
- Some feature packs might not be fully integrated with the WebSphere Application Server product, therefore there are limitations on use of the feature pack functionality.

Documentation on available feature packs can be found in the WebSphere Application Server Version 7.0 Information Center. Using filtering, you can select only those articles that apply to a particular platform, product or feature pack.

Chapter 3. Specifications and API documentation

The WebSphere Application Server product supports various industry standards. This topic lists the specifications and application programming interface (API) documentation supported in current and previous product releases.

Components

- Any application type
- Web applications
- Portlet applications
- SIP applications
- EJB applications
- Client applications
- Web services
- Service Component Architecture
- Service integration
- Data access resources
- Messaging resources
- Mail, URLs, and other Java EE resources
- Security
- Web services security
- Naming and directory
- Object Request Broker
- Transactions
- WebSphere extensions
- Administration

Any application type

Table 1. Supported specifications for any application type

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Platform, Enterprise Edition (Java EE) specification Prior to Java EE 5, the specification name was Java 2 Platform, Enterprise Edition (J2EE).	Java EE 5 New J2EE 1.4 J2EE 1.3	J2EE 1.4 J2EE 1.3 J2EE 1.2	J2EE 1.4 New J2EE 1.3 J2EE 1.2	J2EE 1.3 J2EE 1.2
Java Platform, Standard Edition (Java SE) specification Prior to Java SE 6, the specification name was Java 2 Platform, Standard Edition (J2SE).	Java SE 6 New	J2SE 5	J2SE 1.4.2	J2SE 1.4.2 New J2SE 1.3
ISO 8859 specifications	ISO 8859 applies to these versions.			

Web applications

Table 2. Supported specifications for Web applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Servlet specification (JSR 154 and JSR 53)	Java Servlet 2.5 New Java Servlet 2.4 Java Servlet 2.3	Java Servlet 2.4 Java Servlet 2.3	Java Servlet 2.4 New Java Servlet 2.3	Java Servlet 2.3
JavaServer Pages (JSP) specification (JSR 245, JSR 152, and JSR 53)	JSP 2.1 New JSP 2.0 JSP 1.2	JSP 2.0 JSP 1.2	JSP 2.0 New JSP 1.2	JSP 1.2

Portlet applications

Table 3. Supported specifications for portlet applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Portlet specification	Portlet 2.0 (JSR 286) New	Portlet 1.0 (JSR 168)	Not applicable. The product first supports portlets in Version 6.1.	

Session Initialization Protocol applications

Table 4. Supported specifications and APIs for SIP applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Session Initiation Protocol (SIP) Servlet API (JSR 116) For a complete list of SIP and SIP proxy standards, see SIP industry standards compliance.	SIP 1.0	SIP 1.0	Not applicable. The product first supports SIP in Version 6.1.	

Enterprise bean (EJB) applications

Table 5. Supported specifications and APIs for EJB applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Enterprise JavaBeans (EJB) specification	EJB 3.0 EJB 2.1 EJB 2.0	EJB 3.0 New for Feature Pack for EJB 3.0 EJB 2.1 EJB 2.0 EJB 1.1	EJB 2.1 New EJB 2.0 EJB 1.1	EJB 2.0 EJB 1.1
Java DataBase Connectivity (JDBC) API	JDBC 4.0 New JDBC 3.0 JDBC 2.1 and Optional Package API (2.0)	JDBC 3.0 JDBC 2.1 and Optional Package API (2.0)	JDBC 3.0 New JDBC 2.1 and Optional Package API (2.0)	JDBC 2.0

Table 5. Supported specifications and APIs for EJB applications (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Message Service (JMS) specification	JMS 1.1	JMS 1.1	JMS 1.1 New	JMS 1.02
Java Persistence API (JPA) specification	JPA	JPA	Not applicable	Not applicable

Client applications

Table 6. Supported specifications and APIs for client applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Web Start architecture	Java Web Start 1.4.2	Java Web Start 1.4.2	Java Web Start 1.4.2 New	Not applicable

Web services

Table 7. Supported specifications and APIs for Web services

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Architecture for XML Binding (JAXB) specification	JAXB 2.1 New	JAXB 2.0 New for Feature Pack for Web Services	Not applicable	Not applicable
Java Architecture for XML Binding (JAXB) Reference Implementation Vendor Extensions Runtime Properties specification	JAXB 2.1 RI Vendor Extensions New	JAXB 2.0 RI Vendor Extensions New for Feature Pack for Web Services	Not applicable	Not applicable
Java API for XML Processing (JAXP) specification	1.2 Included in Java SE 6.	1.2 Included in J2SE 5.	1.1 Specification is no longer available.	1.1 Specification is no longer available.
Java API for XML Registries (JAXR) specification	JAXR 1.0	JAXR 1.0	JAXR 1.0 New	Not applicable
Java API for XML-based RPC (JAX-RPC) specification	JAX-RPC 1.1	JAX-RPC 1.1	JAX-RPC 1.1 New	JAX-RPC 1.0
Java API for XML Web Services (JAX-WS) specification	JAX-WS 2.1 New	JAX-WS 2.0 New for Feature Pack for Web Services	Not applicable	Not applicable
Reliable Asynchronous Messaging Profile (RAMP)	RAMP 1.0	RAMP 1.0 New for Feature Pack for Web Services	Not applicable	Not applicable
SOAP	SOAP 1.1 SOAP 1.2	SOAP 1.1 SOAP 1.2 New for Feature Pack for Web Services	SOAP 1.1	SOAP 1.1
SOAP with Attachments API for Java (SAAJ) Specification	SAAJ 1.2 SAAJ 1.3	SAAJ 1.2 SAAJ 1.3 New for Feature Pack for Web Services	SAAJ 1.2 New	SAAJ 1.1

Table 7. Supported specifications and APIs for Web services (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
SOAP over Java Message Service (SOAP over JMS)	W3C SOAP over JMS 1.0 (submission draft)			
SOAP Message Transmission Optimization Mechanism (MTOM)	MTOM 1.0	MTOM 1.0 New for Feature Pack for Web Services	Not applicable	
Streaming API for XML (StAX)	StAX 1.0	StAX 1.0 New for Feature Pack for Web Services	Not applicable	
Universal Description, Discovery and Integration (UDDI)	UDDI 3.0	UDDI 3.0	UDDI 3.0 New	UDDI 2.0
W3C XML Schema	• XML Schema 1.0 • XML Schema Part 1 • XML Schema Part 2	• XML Schema 1.0 • XML Schema Part 1 • XML Schema Part 2	• XML Schema 1.0 • XML Schema Part 1 • XML Schema Part 2	
Web Services Addressing (WS-Addressing)	WS-Addressing 1.0 family of specifications: • 1.0 Core • 1.0 SOAP Binding • 1.0 Metadata	WS-Addressing 1.0 family of specifications: • 1.0 Core • 1.0 SOAP Binding • 1.0 Metadata (partially supported)	Not applicable	
Web Services Atomic Transaction (WS-AT)	WS-AT 1.0 WS-AT 1.1 New	WS-AT 1.0	WS-AT 1.0 New	Not applicable
Web Services Business Activity (WS-BA)	WS-BA 1.0 WS-BA 1.1 New	WS-BA 1.0	Not applicable	
Web Services Coordination (WS-COOR)	WS-COOR 1.0 WS-COOR 1.1 New	WS-COOR 1.0	WS-COOR 1.0 New	Not applicable
Web Services Description Language (WSDL)	WSDL 1.1	WSDL 1.1	WSDL 1.1	WSDL 1.1
Web Services for Java Platform, Enterprise Edition (Java EE) (JSR 109) Prior to Web Services for Java EE, the specification name was Web Services for Java 2 Platform, Enterprise Edition (J2EE).	JSR 109 1.2 New	JSR 109 1.1	JSR 109 1.1 New	JSR 109 1.0

Table 7. Supported specifications and APIs for Web services (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Web Services Interoperability Organization (WS-I) Basic Profile	WS-I Basic Profile 1.1 WS-I Basic Profile 1.2 (draft) WS-I Basic Profile 2.0 (draft)	WS-I Basic Profile 1.1 WS-I Basic Profile 1.2 (draft) New for Feature Pack for Web Services WS-I Basic Profile 2.0 (draft) New for Feature Pack for Web Services	WS-I Basic Profile 1.1 New	WS-I Basic Profile 1.0
Web Services-Interoperability Attachments Profile	WS-I Attachments 1.0	WS-I Attachments 1.0	WS-I Attachments 1.0 New	Not applicable
Web Services Invocation Framework (WSIF)	WSIF	WSIF	WSIF	WSIF
Web Services Metadata for the Java Platform (JSR 181)	Web Services Metadata for the Java Platform	Web Services Metadata for the Java Platform New for Feature Pack for Web Services	Not applicable	
Web Services Notification (WS-Notification)	WS-Notification 1.3 family of specifications: • WS-BaseNotification 1.3 • WS-BrokeredNotification 1.3 • WS-Topics 1.3	WS-Notification 1.3 family of specifications: • WS-BaseNotification 1.3 • WS-BrokeredNotification 1.3 • WS-Topics 1.3	Not applicable	
Web Services Policy (WS-Policy) specification	Web Services Policy 1.5 New Web Services Addressing 1.0 - Metadata New Web Services Atomic Transaction Version 1.0 and Web Services Atomic Transaction Version 1.1 New Web Services Reliable Messaging Policy Assertion Version 1.0 and Web Services Reliable Messaging Policy Assertion Version 1.1 New WS-SecurityPolicy 1.2 New	Not applicable		

Web Services Reliable Messaging	WS-ReliableMessaging 1.0 WS-ReliableMessaging 1.1 WS-MakeConnection Version 1.0 New	WS-ReliableMessaging 1.0 and WS-ReliableMessaging 1.1. New for Feature Pack for Web Services	Not applicable
Web Services Resource Framework (WSRF)	WSRF 1.2	WSRF 1.2 New	Not applicable
XML-binary Optimized Packaging (XOP)	XOP 1.0	XOP 1.0 New for Feature Pack for Web Services	Not applicable

Service Component Architecture

The Feature Pack for Service Component Architecture (SCA) supports the following specifications. The feature pack supports most sections of the specifications, although some sections are not supported. See Chapter 4, "Unsupported SCA specification sections," on page 29.

Table 8. Supported specifications and APIs for SCA applications

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
SCA Assembly Model specification	SCA Assembly Model 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable
SCA Policy Framework specification	SCA Policy Framework 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable
SCA Transaction Policy specification	SCA Transaction Policy 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable
SCA Java Common Annotations and APIs specification	SCA Java Common Annotations and APIs 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable
SCA Java Component Implementation specification	SCA Java Component Implementation 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable
SCA Web service binding specification	SCA Web service binding 1.00 New for Feature Pack for SCA Version 1.0.0	Not applicable	Not applicable	Not applicable

Table 8. Supported specifications and APIs for SCA applications (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
SCA EJB Session Bean Binding specification	SCA EJB Session Bean Binding 1.00 New for Feature Pack for SCA Version 1.0.0 Supports EJB 2.1 and 3.0 modules.	Not applicable	Not applicable	Not applicable

Service integration

Table 9. Supported specifications and APIs for service integration

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java DataBase Connectivity (JDBC) API	JDBC 4.0 New	JDBC 3.0	JDBC 3.0 New	JDBC 2.0

Data access resources

Table 10. Supported specifications and APIs for data access resources

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java DataBase Connectivity (JDBC) API	JDBC 4.0 New	JDBC 3.0	JDBC 3.0 New	JDBC 2.0
Java EE Connector Architecture (JCA) resource adapter	JCA 1.5	JCA 1.5	JCA 1.5 New	JCA 1.0
Service Data Objects (SDO) specification	SDO 1.0	SDO 1.0	SDO 1.0 New	Not applicable

Messaging resources

Table 11. Supported specifications and APIs for messaging resources

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Message Service (JMS)	JMS 1.1	JMS 1.1	JMS 1.1 New	JMS 1.0.2
Java EE Connector Architecture (JCA) resource adapter	JCA 1.5	JCA 1.5	JCA 1.5 New	JCA 1.0

Mail, URLs, and other Java EE resources

Table 12. Supported specifications and APIs for mail, URLs, and other Java EE resources

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
JavaMail API documentation (JSR 919)	JavaMail 1.4 New	JavaMail 1.3	JavaMail 1.3 New	JavaMail 1.2
URL API documentation	URL 1.4.2	URL 1.4.2	URL 1.4.2 New	1.2 Specification is no longer available.
JavaBeans Activation Framework (JAF) Specification	JAF 1.1 New	JAF 1.0.2	JAF 1.0.2 New	JAF 1.0

Table 12. Supported specifications and APIs for mail, URLs, and other Java EE resources (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
W3C Architecture - Naming and Addressing: URIs, URLs	W3C Naming and Addressing applies to these versions.			

Security

Table 13. Supported specifications and APIs for security

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java 2 Security Manager	Java 2 Security Manager 1.5	Java 2 Security Manager 1.5	Java 2 Security Manager 1.4 New	Java 2 Security Manager 1.3
Java Authentication and Authorization Service (JAAS)	JAAS 2.0 applies to these versions.			
Java Authorization Contract for Containers (JACC)	JACC 1.1 New	JACC 1.0	JACC 1.0 New	Not applicable
Common Secure Interoperability Version 2 (CSIv2) specification This is an Object Management Group (OMG) CORBA/IIOP specification.	CSI 2.0 applies to these versions.			
Secure Sockets Layer (SSL) configuration The product uses Java Secure Sockets Extension (JSSE) as the SSL implementation for secure connections. JSSE is part of the Java 2 Standard Edition (J2SE) specification and is included in the IBM implementation of the Java Runtime Extension (JRE) specification.	JSSE 5.0	JSSE 5.0 New	JSSE 1.0.3	JSSE 1.0.3
Java Generic Security Service (JGSS) Use JGSS with the Kerberos Network Authentication Service, Version 5	JGSS 1.0.1 applies to these versions.			
The Simple and Protected GSS-API Negotiation Mechanism (SPNEGO)	SPNEGO 1.0 applies to these versions.			
Java Cryptographic Extension (JCE) specification	JCE 1.0 applies to these versions.			
Java Certification Path (CertPath) API	CertPath 1.1	CertPath 1.1 New CertPath 1.0	CertPath 1.0	CertPath 1.0

Web services security

Table 14. Supported specifications and APIs for Web services security

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Canonical XML	Canonical XML 1.0 applies to these versions.			
Decryption Transform for XML Signature	Decryption Transformation for XML Signature applies to these versions..			
Exclusive XML Canonicalization	Exclusive XML Canonicalization 1.0 applies to these versions.			

Table 14. Supported specifications and APIs for Web services security (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
OASIS Web Services Security: SOAP Message Security (WS-Security)	WS-Security 1.0 WS-Security 1.1	WS-Security 1.0 WS-Security 1.1 New for Feature Pack for Web Services	WS-Security 1.0	WS-Security draft 13
OASIS Web Services Security: Kerberos Token Profile	Kerberos Token Profile 1.1 New	Not applicable		
OASIS Web Services Security: Username Token Profile	Username Token Profile 1.0 Username Token Profile 1.1	Username Token Profile 1.0 Username Token Profile 1.1 New for Feature Pack for Web Services	Username Token Profile 1.0 New	Username Token Profile Draft 2
OASIS Web Services Security: X.509 Token Profile	X.509 Token Profile 1.0 X.509 Token Profile 1.1	X.509 Token Profile 1.0 X.509 Token Profile 1.1 New for Feature Pack for Web Services	X.509 Token Profile 1.0 New	Not applicable
Web Services Interoperability Organization (WS-I) Basic Security Profile	WS-I Basic Security Profile 1.0 WS-I Basic Security Profile 1.1 New	WS-I Basic Security Profile 1.0	Not applicable	
Web Services Interoperability Organization (WS-I) Reliable Secure Profile	WS-I Reliable Secure Profile 1.0 (draft)	WS-I Reliable Secure Profile 1.0 (draft) New for Feature Pack for Web Services	Not applicable	
Web Services Secure Conversation (WS-SecureConversation)	OASIS WS-SecureConversation 1.0 (submission draft) OASIS WS-SecureConversation 1.3 New	OASIS WS-SecureConversation 1.0 (draft submission) New for Feature Pack for Web Services	Not applicable	

Table 14. Supported specifications and APIs for Web services security (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Web Services Trust	OASIS WS-Trust 1.1(submission draft) OASIS WS-Trust 1.3 New	OASIS WS-Trust 1.1 New for Feature Pack for Web Services	Not applicable	
XML Signature Syntax and Processing	XML Signature Syntax and Processing applies to these versions.			
XML Encryption Syntax and Processing	XML Encryption Syntax and Processing applies to these versions.			

Naming and directory

Table 15. Supported specifications and APIs for naming and directory

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Naming and Directory Interface (JNDI) Specification See also JNDI support.	JNDI on Java SE 6 New	JNDI on J2SE applies to these versions.		
Common Object Request Broker: Architecture and Specification (CORBA) specification This is an Object Management Group (OMG) Interoperable Naming (CosNaming) specification.	CORBA 2.4 applies to these versions.			
Interoperable Naming Service specification This is an OMG CosNaming specification.	Interoperable Naming Service			
Naming Service specification This is an OMG CosNaming specification.	Naming Service applies to these versions.			

Object Request Broker

The Object Request Broker (ORB) component follows the Common Object Request Broker Architecture (CORBA) specifications supported by Java 2 Platform, Standard Edition (J2SE). The Object Management Group (OMG) produces the specifications.

Versions 6.1 and later use the J2SE 5.0 specifications that are listed in *Official Specifications for CORBA support in J2SE 5.0* at http://java.sun.com/j2se/1.5.0/docs/guide/idl/compliance.html.

Versions 5.1.x and 6.0.x use the J2SE 1.4 specifications that are listed in *Official Specifications for CORBA support in J2SE 1.4* at http://java.sun.com/j2se/1.4.2/docs/api/org/omg/CORBA/doc-files/compliance.html.

Table 16. Supported specifications and APIs for ORB

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Common Object Request Broker Architecture (CORBA) specifications	CORBA 2.3.1 applies to these versions.			
Revised IDL to Java language mapping	Revised IDL to Java language mapping applies to these versions.			
New IDL to Java Mapping Chapter	New IDL to Java Mapping Chapter applies to these versions.			
Updated Java to IDL Mapping specification	Updated Java to IDL Mapping applies to these versions.			
Interoperable Naming Service revised chapters	Interoperable Naming Service revised chapters applies to these versions.			
Object Reference Template Final Adopted specification	Object Reference Template Final Adopted specification	Object Reference Template Final Adopted specification New	Not applicable	
Portable Interceptors specification	Not applicable	Not applicable	Portable Interceptors specification	

Transactions

Table 17. Supported specifications and APIs for transactions

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
CORBA Object Transaction Service (OTS) specification	OTS 1.4	OTS 1.4	OTS 1.4 New	OTS 1.2
Java EE Connector Architecture (JCA) resource adapter	JCA 1.5	JCA 1.5	JCA 1.5 New	JCA 1.0
Java Transaction API (JTA) specification	JTA 1.1 New	JTA 1.0.1B	JTA 1.0.1B New	JTA 1.0.1
Java Transaction Service (JTS) specification	JTS 1.0 applies to these versions.			
Web Services Atomic Transaction (WS-AT)	WS-AT 1.0 WS-AT 1.1 New	WS-AT 1.0	WS-AT 1.0 New	Not applicable
Web Services Business Activity (WS-BA)	WS-BA 1.0 WS-BA 1.1 New	WS-BA 1.0	Not applicable	
Web Services Coordination (WS-COOR)	WS-COOR 1.0 WS-COOR 1.1 New	WS-COOR 1.0	WS-COOR 1.0 New	Not applicable

WebSphere extensions

Table 18. Supported specifications and APIs for WebSphere extensions

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
ActivitySession service and Last Participant Support				
J2EE Activity Service for Extended Transactions (JSR 95)	JSR 95 applies to these versions.			

Table 18. Supported specifications and APIs for WebSphere extensions (continued)

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java Transaction API (JTA) specification	JTA 1.1 New	JTA 1.0.1B New	JTA 1.0.1	JTA 1.0.1
Internationalization (i18n)				
J2SE internationalization documentation	J2SE Internationalization 5.0	J2SE Internationalization 5.0 New	J2SE Internationalization 1.4.2	J2SE Internationalization 1.4.2

Administration

Table 19. Supported specifications and APIs for administration

Specification or API	Version 7.0	Version 6.1	Version 6.0	Version 5.1
Java EE Application Deployment specification	Java EE Deployment 1.2 New	J2EE Deployment 1.1	J2EE Deployment 1.1 New	Not applicable
J2EE Extension Mechanism Architecture	J2EE Extension Mechanism Architecture 1.4.2	J2EE Extension Mechanism Architecture 1.4.2	J2EE Extension Mechanism Architecture 1.4.2 New	Not applicable
Java Management Extensions (JMX) JSR-000003	JMX 1.2	JMX 1.2	JMX 1.2 New	JMX 1.0
Java Management Extensions (JMX) Remote API	JMX Remote API 1.0	JMX Remote API 1.0 New	Not applicable	
Java Virtual Machine (JVM) specification See WebSphere Application Server detailed system requirements.	JVM 6 New	JVM 5.0 New	JVM 1.4.2	JVM 1.4.2
Logging API specification (JSR 47)	Logging API specification (JSR 47) 1.0	Logging API specification (JSR 47) 1.0	Logging API specification (JSR 47) 1.0 New	Not applicable

Related information

WebSphere Application Server detailed system requirements

Chapter 4. Unsupported SCA specification sections

This topic lists the sections of Service Component Architecture (SCA) specifications not supported in Feature Pack for SCA.

- SCA Assembly Model
- SCA Policy Framework
- SCA Transaction Policy
- SCA Java Common Annotations and APIs
- SCA Web Service Binding
- SCA EJB Session Bean Binding

The following tables list the unsupported sections of the indicated SCA specifications. The SCA Java Component Implementation specification is fully supported, so does not have a list of unsupported sections.

SCA Assembly Model

Table 20. Unsupported sections of the SCA Assembly Model specification

Section	Not supported in Feature Pack for SCA v1.0
1.3 Component	• Component attribute: constrainingType • A component element has zero implementation elements • Reference attribute wiredByImpl
1.4.1 Component Type	constrainingType
1.5 Interface	WSDL 2.0 interfaces are not supported.
1.5.2 Bidirectional Interfaces	Callback is not supported for EJB binding.
1.5.3 Conversational Interfaces	Conversation is not supported.
1.5.4 SCA-Specific Aspects for WSDL Interfaces	Conversation is not supported.
1.6 Composite	• Composite Attribute: local (optional) – whether all the components within the composite must all run in the same operating system process. `local="true"` means that all the components must run in the same process. `local="false"`, which is the default, means that different components within the composite might run in different operating system processes. Feature Pack for SCA behavior is that, local or not, all components within the composite are deployed on the same Java virtual machine (JVM). • constrainingType

Table 20. Unsupported sections of the SCA Assembly Model specification (continued)

Section	Not supported in Feature Pack for SCA v1.0
1.6.2 Reference	• Composite reference attribute: wiredByImpl • Autowire only supported for the components within the same composite • The bindings defined on the component reference are still in effect for local wires within the composite that have the component reference as their source. Feature Pack for SCA limits the function wiring reference to outside service using binding specific endpoint URI (or using reference target). Wiring to local componentService is only supported for default binding.
1.6.3 Service	The bindings defined on the component service are still in effect for local wires within the composite that target the component service. Feature Pack for SCA limits this function. Local component service can only be wired through default binding from a local component reference.
1.6.4 Wire	Wire is not supported.
1.6.8 ConstrainingType	ConstrainingType
1.7.2.1 Constructing Hierarchical URIs	For the default binding, the Feature Pack for SCA does not support the @uri attribute on the service-side binding. In other words, using a non-default URI on a service exposed over the default binding is not supported. Specifically, the @uri attribute should not be used on a <binding.sca> element that is a child of a component <service> element.
1.10.2 Contributions	OSGi bundle as contribution
1.10.2.2 SCA Contribution Metadata Document	sca-contribution-generated.xml
1.10.4.2 add Deployment Composite & update Deployment Composite	Update Deployment Composite.supported through the business-level application updateAssets.command.

SCA Policy Framework

Table 21. Unsupported sections of the SCA Policy Framework specification

Section	Not supported in Feature Pack for SCA v1.0
1 Policy Framework	• `@policySets` (except for authorization policy) • `definitions.xml` (except for authorization policy) • callbacks • `<operation>` element • componentType file
1.9 Miscellaneous Intents	The following miscellaneous intents are not supported: • SOAP • JMS • NoListener • BP.1_1

SCA Transaction Policy

Table 22. Unsupported sections of the SCA Transaction Policy specification

Section	Not supported in Feature Pack for SCA v1.0
	`<operation>` element

SCA Java Common Annotations and APIs

Table 23. Unsupported sections of the SCA Java Common Annotations and APIs specification

Section	Not supported in Feature Pack for SCA v1.0
1 Common Annotations, APIs, Client and Implementation Model	Conversation is not supported
1.2.4.2 - 1.2.4.3 Implementation Metadata	`@SCOPE("COMPOSITE")` is not supported in the clustered environment, which means that ″All service requests are dispatched to the same implementation instance for the lifetime of the containing composite″ is not supported in the clustered environment.

SCA Web Service Binding

Table 24. Unsupported sections of the SCA Web service binding specification

Section	Not supported in Feature Pack for SCA v1.0
2.1 Web Service Binding Schema	• Line 47: `wsdl.endpoint` is not supported in `/binding.ws/@wsdlElement` • Line 55: `wsdlLocation` is not supported in `/binding.ws/@wsdli:wsdlLocation`

2.1.1 Endpoint URI resolution	• Lines 71-79: Ordering of implementation in Feature Pack for SCA is shown below:
	`ordering for reference side: reference target-> location in wsdl -> EndPointReference -> binding.ws uri`
	`ordering for service side: binding.ws name -> binding.ws uri -> implicit (component/service)`
	• Line 73: URI in referenced WSDL (support limited to reference side)
	• Line 76: Explicit URI in `binding.ws` (support limited to reference side as absolute URI, on service side as relative URI (`contextRoot`))
	• Line 78: Implicit URI in `binding.ws` (support limited to service only)

SCA EJB Session Bean Binding

Table 25. Unsupported sections of the SCA EJB Session Bean Binding specification

Section	Not supported in Feature Pack for SCA v1.0

2.1 Session Bean Binding Schema	`/binding.ejb/@session-type`
	• Since Feature Pack for SCA does not support conversations, although session-type is set to ″stateful″, the service still behaves as stateless.
	`/binding.ejb/@uri`
	• Line 91: Feature Pack for SCA only supports the following formats:
	– For EJB2
	`corbaname:iiop:<hostName>:<port>/ NameServiceServerRoot#ejb/ sca/ejbbinding/<componentName>/<serviceName>`
	– For EJB3
	`corbaname:iiop:<hostName>: <port>/NameServiceServerRoot#ejb/sca/ejbbinding/ <componentName>/<serviceName># <serviceInterfaceName>Remote or corbaname:iiop: <hostName>: <port>/NameServiceServerRoot# <serviceInterfaceName>Remote`
	• Line 97: corbaname:rir:#ejb/MyHome

2.3.1 Conversational Nature of Stateful Session Beans	Lines 197-229

Chapter 5. Developing Service Component Architecture (SCA) services and applications

Developing SCA services from existing WSDL files

You can develop an Service Component Architecture (SCA) service implementation when starting with an existing Web Services Description Language (WSDL) file.

Before you begin

Locate the WSDL file that defines the SCA service that you want to implement. You can develop a WSDL file or obtain one from an existing SCA service. The WSDL file describes your service interface as a WSDL portType and includes XSD schema definitions of your business data.

About this task

There are two ways to develop an SCA service implementation:

- Top-down development starting with an existing Web Services Description Language (WSDL)
- Bottom-up development starting from existing Java code that uses Java Architecture for XML Binding (JAXB) data types

The top-down development approach takes advantage of the interoperable XML-based WSDL, and XSD interface and data definitions.

This task describes the steps when using the top-down development approach to develop an SCA service implementation in Java when starting from a WSDL interface and XSD data definitions.

Note: It is a best practice to use the top-down methodology to develop SCA service implementations because this approach leverages the capabilities of the XML interface description and provides greater ease in interoperability across platforms, bindings, and programming languages.

Use the wsimport command-line tool to generate the Java representations of your business service interfaces and your business data when an existing WSDL file describes the desired SCA service interface as a WSDL portType, along with XSD schema definitions of your business data. The wsimport tool generates Java classes that you can use to write a Java implementation that reflects your business logic. The result is a Plain Old Java Object (POJO) implementation of the generated interface using the generated JAXB data types. By adding the @Service annotation to the Java implementation, the annotation defines the Java implementation as an SCA service implementation.

The generated annotated Java classes that correspond to your business data contain all the necessary information that the JAXB runtime environment requires to build and parse the XML for marshaling and unmarshaling. In other words, the data programming model is limited to object instantiation and the use of getter and setter methods, and you do not need to write code to convert the data between the XML wire format and the Java application.

Note: The Feature Pack for SCA uses XML marshalling as defined by JAXB to marshal and unmarshal data across a remotable interface. If you start with a remotable Java interface for your implementation rather than starting with a WSDL portType interface, be careful when selecting the input and output Java data types and ensure you understand which data is preserved across JAXB marshalling and unmarshalling. However, when authoring an implementation on a local interface, you can use any Java type because local interfaces use pass-by-reference semantics, which implies no data is copied.

Note: The Feature Pack for Service Component Architecture (SCA) does not support using a WSDL file when the Java mapping requires holder classes. The Feature Pack for SCA uses the JAX-WS specification to define the mapping between WSDL files and Java, including the mapping between a WSDL portType object and a Java interface. When you have WSDL portType objects with operations that use in-out parameters or operations that use multiple output parameters, the JAX-WS specification uses instances of the javax.xml.ws.Holder class in the mapping of the WSDL portType object to a Java interface. When using the Feature Pack for SCA, do not use a WSDL file when the Java mapping requires holder classes. Instead, use a WSDL file that does not map to holder classes.

When you develop an SCA service when starting from an existing WSDL file, the interface is considered a remotable interface. The remotable interface uses pass-by-value semantics, which implies your data is copied.

You can use and deploy the resulting Java implementation as an SCA component that is defined in a composite definition. The composite definition defines SCA artifacts, such as service references, imports, and exports. The component is defined in terms of development artifacts such as the WSDL, the Java implementation, and bindings that are defined during deployment.

1. Use the wsimport command-line tool to develop SCA Java representations of your business service interfaces and your business data.

   The wsimport tool processes a WSDL file and generates Java classes and the JAXB data types that are used to create the SCA service.

   It is important to include all generated classes within your application archive, including the classes that you might not directly reference in your Java implementation. Even for the case where you have simple interfaces that pass simple parameter types like String and Integer, or where no JAXB data types are necessary, be sure to include all classes, including indirect references, in this code generation step.

   - Run the wsimport command to generate the artifacts.

   The wsimport tool is located in the *app_server_root*\bin\ directory.

   ```
   app_server_root\bin\wsimport.bat -keep wsdl_URL
   ```

   ```
   app_server_root/bin/wsimport.sh -keep wsdl_URL
   ```

   ```
   app_server_root/bin/wsimport -keep wsdl_URL
   ```

   The **-keep** option specifies to keep the generated Java source files and the compiled class files.

2. Locate the Java interface that directly corresponds to your WSDL portType from the generated artifacts. The interface is generated with an @WebService annotation, and it is an interface and not a class file.
3. Complete the implementation of your SCA service. Write a Java implementation of the generated Java interface that reflects your business logic. The Java implementation is a Plain Old Java Object (POJO) implementation of the generated interface using the generated JAXB data types. This implementation is annotated based on the SCA Java component implementation programming model.
4. Annotate the Java implementation. Add the @Service annotation to the Java implementation to specify this is an SCA service. When you complete this step, you have created an SCA component implementation.
5. Define a component within a composite definition using this component implementation. In the definition of your composite, define a component that refers back to the original WSDL portType interface and the SCA implementation.
   a. Under the `<component>` element, create a `<implementation.java>` child element that refers to the class name of your POJO component implementation.
   b. Under the `<component>` element, create a `<service>` child element.
   c. Under the `<service>` element, create a `<interface.wsdl ..>` element that refers back to the WSDL portType. The @name attribute of the `<service>` element must match the unqualified class name of your Java interface.

   You now have a component with a well-defined component name and service name with a well-defined interface.

   In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the `<implementation.java>`, `<interface.wsdl>` and `<service>` elements described in this step.
6. Deploy the SCA service by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have created an SCA implementation by starting with an existing WSDL file.

Example

The following example illustrates using an existing WSDL interface to generate a Java interface that is used to create a Java implementation that is an SCA service.

1. Copy the following sample `account.wsdl` WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:account="http://www.myaccount.com/account"
```

```
targetNamespace="http://www.myaccount.com/account"
name="AccountService">

 <wsdl:types>
    <schema targetNamespace="http://www.myaccount.com/account"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:account="http://www.myaccount.com/account">

          <element name="computeAccountAverage">
            <complexType>
                <sequence>
                    <element name="account" type="account:Account" />
                    <element name="days" type="xsd:int" />
                </sequence>
            </complexType>
        </element>
        <element name="computeAccountAverageResponse">
            <complexType>
                <sequence>
                    <element name="return" type="xsd:float" />
                </sequence>
            </complexType>
        </element>

        <complexType name="Account">
            <attribute name="accountNumber" type="xsd:int" />
            <attribute name="accountID" type="xsd:string" />
            <attribute name="accountType" type="xsd:string" />
            <attribute name="balance" type="xsd:float" />
        </complexType>

     </schema>
</wsdl:types>

<wsdl:message name="computeAccountAverageRequest">
    <wsdl:part element="account:computeAccountAverage"
        name="parameters" />
</wsdl:message>

<wsdl:message name="computeAccountAverageResponse">
    <wsdl:part element="account:computeAccountAverageResponse"
        name="parameters" />
</wsdl:message>

<wsdl:portType name="AccountService">
    <wsdl:operation name="computeAccountAverage">
        <wsdl:input message="account:computeAccountAverageRequest" name="accountReq"/>
        <wsdl:output message="account:computeAccountAverageResponse" name="accountResp"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AccountServiceSOAP" type="account:AccountService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="computeAccountAverage">
       <soap:operation
            soapAction="computeAccountAverage" />
        <wsdl:input name="accountReq">
           <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output name="accountResp">
            <soap:body use="literal" />
        </wsdl:output>
    </wsdl:operation>

 </wsdl:binding>
```

```
    <wsdl:service name="AccountWSDLService">
        <wsdl:port binding="account:AccountServiceSOAP"
            name="AccountServicePort">
            <soap:address location=""/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

2. Run the wsimport command from the *app_server_root*\bin\ directory.

   ```
   app_server_root\bin\wsimport.bat -keep -verbose account.wsdl
   ```

   Run the wsimport command,

   ```
   app_server_root/bin/wsimport.sh -keep -verbose account.wsdl
   ```

   ```
   app_server_root/bin/wsimport -keep -verbose account.wsdl
   ```

   After generating the template files using the wsimport command, the following Java files are generated:

   ```
   com/myaccount/account/Account.java
   com/myaccount/account/AccountService.java
   com/myaccount/account/AccountWSDLService.java
   com/myaccount/account/ComputeAccountAverage.java
   com/myaccount/account/ComputeAccountAverageResponse.java
   com/myaccount/account/ObjectFactory.java
   com/myaccount/account/package-info.java
   ```

3. Identify the generated Java interface from the generated classes.

```
//
// Generated By:JAX-WS RI IBM 2.1.1 in JDK 6 (JAXB RI IBM JAXB 2.1.3 in JDK 1.6)
//
package com.myaccount.account;
...
@WebService(name = "AccountService", targetNamespace = "http://www.myaccount.com/account")
...
public interface AccountService {
    /**
     *
     * @param days
     * @param account
     * @return
     *     returns float
     */
    @WebMethod(action = "computeAccountAverage")
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "computeAccountAverage", targetNamespace = "http://www.myaccount.com/account",
            className = "com.myaccount.account.ComputeAccountAverage")

    @ResponseWrapper(localName = "computeAccountAverageResponse", targetNamespace = "http://www.myaccount.com/account",
            className = "com.myaccount.account.ComputeAccountAverageResponse")
    public float computeAccountAverage(
        @WebParam(name = "account", targetNamespace = "")
        Account account,
        @WebParam(name = "days", targetNamespace = "")
        int days);

}
```

   This code example is a Java interface, not merely a Java class. The @WebService annotation is present in this Java interface. It is important to know that this example is not the same as the generated @WebServiceClient class, com.myaccount.account.AccountWSDLService, which is not an interface and is not needed in your SCA application.

4. Complete the implementation of your SCA service by writing a Java implementation of this generated Java interface. Be sure to add the SCA @Service annotation to the implementation.

```
package com.myaccount.account;
import org.osoa.sca.annotations.Service;
@Service(AccountService.class)
public class AccountServiceImpl implements AccountService
    public float computeAccountAverage( Account account, int days) {

        // Write your business logic here.  Account is a
        // generated JAXB type and so use the JAXB programming model.
        // For example, object instantation is performed using
        // the ObjectFactory.createAccount()) method.
    }
}
```

By completing this step, you have completed a component implementation. Not only is this a Java implementation of a Java interface, but the @Service annotation signifies that this is a Java component implementation of an SCA service interface. The implementation class itself does not need all the JAX-WS or JAXB annotations. The runtime environment loads the appropriate annotations from the generated classes that the implementation refers to.

5. Create a component using the component implementation. You create a component definition in a composite that references the original WSDL portType interface and the SCA implementation. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://account.customer"
         name="accountComposite">
     <component name="BankingComponent">
         <implementation.java class="com.myaccount.account.AccountServiceImpl"/>

         <!-- The @name value matches the contents of the @Service which in turn
              comes from the WSDL portType. -->

         <service name="AccountService">

              <!-- This statement specifies the QName of the WSDL portType,
                    "http://www.myaccount.com/account#AccountService" in the syntax as
                    illustrated in the interface.wsdl statement. -->

             <interface.wsdl interface="http://www.myaccount.com/account#wsdl.interface(AccountService)" />
             <binding.ws/>

  <!-- This example uses the SCA Web service binding.  However, it does not matter which SCA binding
       you choose.  -->

          </service>
     </component>
</composite>
```

6. After your component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are ready to deploy the SCA service by creating an SCA business level application.

Developing SCA services with existing Java code

You can develop an Service Component Architecture (SCA) service implementation when starting with an existing Java application.

About this task

There are two ways to develop an SCA service implementation:

- Top-down development starting with an existing Web Services Description Language (WSDL)
- Bottom-up development starting from existing Java code that uses Java Architecture for XML Binding (JAXB) data types

The bottom-up development approach provides a simplified way to begin developing SCA services for the Java developer that does not desire to work with WSDL or XML schema (XSD) authoring or when building new SCA services that expose existing legacy implementations with Java interfaces.

The top-down development approach takes advantage of the interoperable XML-based WSDL, and XSD interface and data definitions.

This task describes the steps when using the bottom-up development approach to develop an SCA service implementation when starting with Java.

When using the bottom-up development methodology, begin by writing a Java interface and implementation that describes the desired business logic. This implementation is then packaged into an application archive file such as a Web application archive (WAR) file or a Java archive (JAR) file that is subsequently is used by an SCA component that is configured with deployment information containing the bindings when the SCA application is deployed.

Note: It is a best practice to use the top-down methodology to develop SCA service implementations because this approach leverages the capabilities of the XML interface description and provides a greater ease in interoperability across platforms, bindings, and programming languages. To learn more about using the top-down methodology, read about developing SCA services from existing WSDL files.

Note: The Feature Pack for SCA uses XML marshalling as defined by JAXB to marshal and unmarshal data across a remotable interface. If you start with a remotable Java interface for your implementation rather than starting with a WSDL portType interface, be careful when selecting the input and output Java data types and ensure you understand which data is preserved across JAXB marshalling and unmarshalling. However, when authoring an implementation on a local interface, you can use any Java type because local interfaces use pass-by-reference semantics, which implies no data is copied.

Note: The data marshalling and unmarshalling that is used to instantiate the copying of data over remotable interfaces is defined by the JAXB specification rather than by Java serialization or the java.io.Serializable or java.io.Externalizable interfaces. Because of this behavior, certain existing Java types are not suitable for use on remotable interfaces, as these types are not serialized using Java serialization. For data types that are not annotated, the class is introspected and its Java properties determine the data that is preserved in the copy. For data types that take advantage of JAXB annotations, you can customize the mapping of Java classes to XSD types and of Java instances to XML documents. Custom Java serialization routines such as the readObject() or writeObject() are not applicable in this scenario. The SCA runtime environment takes an XML centric view of the business

data and leverages the JAXB standards to define the mappings between the Java programming model and the XML data format on the wire.

1. Access the existing Java interface that you want to expose as an SCA service.
2. Determine if you are using a local or a remotable interface.
   - If you are using a remotable interface, add the @Remotable annotation to the Java interface. The input and output Java data types on the remotable interface use pass-by-value semantics which implies your data is copied using XML serialization as defined by JAXB.
3. Complete the implementation of your SCA service. Write a Java implementation of the generated Java interface that reflects your business logic. The Java implementation is a Plain Old Java Object (POJO) implementation of the original interface.
4. Annotate the Java implementation. Add the @Service annotation to the Java implementation to specify this is an SCA service. When you complete this step, you have created an SCA component implementation.
5. Define a component within a composite definition using this component implementation. In the deinition of your composite, define a component that refers back to the original Java interface and the SCA implementation.
   a. Under the `<component>` element, create a `<implementation.java>` child element that refers to the class name of your POJO component implementation.
   b. Under the `<component>` element, create a `<service>` child element.
   c. Under the `<service>` element, create a `<interface.java ..>` element that refers back to the original Java interface. The @name attribute of the `<service>` element must match the unqualified class name of your Java interface.

   You now have a component with a well-defined component name and service name with a well-defined interface.

   In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the `<implementation.java>`, `<imterface.java>` and `<service>` elements described in this step.
6. Deploy the SCA service by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have developed an SCA service using the bottom-up methodology by starting with an existing Java interface or implementation.

Example

The following example illustrates how to create a component implementation of a remotable SCA service interface starting from existing Java code:

1. Start with Java interface `myintf.NameGetter` using type `mypkg.Person`.

```
//NameGetter.java
package myintf;
import mypkg.Person;
public interface NameGetter {
    public String getName(Person p);
}

//Person.java

package mypkg;

public class Person {

    protected String firstName;
    protected String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String value) {
        this.firstName = value;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String value) {
        this.lastName = value;
    }
}
```

   In this example, the mypkg.Person class is well-suited for use over a remotable interface, because it follows the JavaBeans pattern and contains a public getter and setter pair for its important data fields. The XML wire format used by the runtime environment will serialize and deserialize this class. However, other existing Java types that do not adhere to the JavaBeans pattern can cause problems as they dol not serialize correctly and data loss occurs. For this reason, it is a best practice to use a top-down development approach, starting from schema definitions and generating JAXB classes for use in the application programming model. See the developing SCA services from existing WSDL files to learn more about the top-down development approach.

2. Because we are creating a service with a remotable interface, add the @Remotable annotation.

```
//NameGetter.java
package myintf;

import mypkg.Person;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface NameGetter {
    public String getName(Person p);
}
```

3. Unless you have an existing Java implementation, write a Java implementation of the generated Java interface that reflects your business logic.

```
package myintf;
import mypkg.Person;

public class NameGetterImpl implements NameGetter {
```

```
    public String getName(Person p) {
        // Example "business logic"
        return p.getFirstName() + " " + p.getLastName();
    }

}
```

4. Add the @Service annotation to the Java implementation.

```
package myintf;
import mypkg.Person;
import org.osoa.sca.annotations.Service;

@Service(NameGetter.class)
public class NameGetterImpl implements NameGetter {

    public String getName(Person p) {
        // Example "business logic"
        return p.getFirstName() + " " + p.getLastName();
    }

}
```

5. Create a component using the component implementation. You will create a component definition in a composite that references the original Java implementation class, as well as its Java interface. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://org.services.naming"
        name="NameServices">

      <component name="NamingServicesComponent">
         <implementation.java class="myintf.NameGetterImpl"/>
         <service name="NameGetter">

             <!-- The interface.java is not required because the run time can introspect it. -->

             <interface.java interface="myintf.NameGetter"/>

             <!-- The choice of bindings is not important for the example.  Here, both the SCA default and
                  Web services bindings are configured. -->
             <binding.ws/>
             <binding.sca/>
         </service>
     </component>

</composite>
```

6. After your component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are ready to deploy the SCA service by creating an SCA business level application.

Developing SCA service clients

You can develop an Service Component Architecture (SCA) service client starting with either a Java interface or a WSDL file for the SCA service that you want to invoke.

About this task

You can develop SCA service clients that can both access and invoke an SCA service that are based on the Service Component Architecture specification. An SCA client can consume a diverse set of services such as enterprise beans, Web services, and other SCA services, through the capabilities of the respective SCA bindings and by using the Plain Old Java Object (POJO) client programming model.

To develop SCA service clients, you can start with either an existing Web Services Description Language (WSDL) file and use the wsimport tool to generate the Java interface or you can start with an existing Java interface.

Developing SCA client components starting with an existing WSDL file

When you have an existing WSDL file that describes your SCA service interface as a WSDL portType, along with XSD schema definitions of your business data, you can use the wsimport tool to generate the SCA Java representations of your business service interfaces and your business data. The wsimport tool generates Java classes that you can use to write a Java implementation that reflects your business logic. You can use the generated output of the proxy class and the JAXB data binding types in your Java client to invoke the SCA service using the simple POJO programming model.

The generated annotated Java classes that correspond to your business data contain all the necessary information that the Java Architecture for XML Binding (JAXB) runtime environment requires to build and parse the XML for marshaling and unmarshaling. In other words, the data programming model is limited to object instantiation and the use of getter and setter methods, and you do not need to write code to convert the data between the XML wire format and the Java application.

Now that you have the generated annotated Java classes, you must use the Java interface and data type classes to create the reference proxy as described in the developing SCA clients starting with a Java interface section.

Developing SCA client components starting with a Java interface

When you have a Java interface to your SCA service, obtained either by starting from a WSDL and generating the Java classes or by starting with Java code, use the Java interface and data type classes to create the reference proxy. If your client is designed so that its reference proxy is injected from the SCA container, the Java interface is the same type as your proxy field and this file contains the corresponding @Reference annotation. You can only create the static reference from another SCA component implementation that acts as a client of the original service. If your reference proxy is created programmatically, you must create a proxy variable that has the same type as your Java interface, and use an API such as `CompositeContext.getService(Class <B>interfaze, ...)` to create the reference proxy. The generated Java interface type is the interface parameter that is passed to this API. Read about locating and invoking SCA services to learn more about creating the reference proxy dynamically.

Regardless of whether the proxy is created by injection methods or programmatically, the Java interface is the class of the proxy and the generated JAXB types are the parameter types which includes inputs, outputs, and exceptions.

Considerations for local and remotable interfaces

It is important to understand that a remotable interface uses an XML wire format for data. Therefore, clients must use a JAXB-based programming model for the data types. In contrast, a local interface uses pass-by-reference semantics, so there is no data copy. Using the local interface, data is read and written without any special programming model such as JAXB.

Though WSDL-based interfaces are always remotable, you can also mark a Java interface that is not generated from a WSDL file as remotable by annotating it with the @Remotable annotation. The @Remotable annotation results in a data copy with XML serialization as defined by JAXB.

Defining the remotable interface is straightforward when you start with a WSDL interface, because you use the wsimport tool to generate the JAXB data types that you use when you write your SCA client. The remotable interface is less apparent when starting from a remotable Java interface, unless the Java types are decorated with JAXB annotations. XML serialization behaves differently than Java serialization. For an POJO that is not annotated, Java serialization preserves instance data including private fields, whereas JAXB serialization preserves JavaBeans properties.

The focus of this topic is the use of remotable interfaces.

You can develop a component that consumes or acts as a client of the target service using a component reference. In addition to consuming a service from another component's reference, the Feature Pack for SCA also provides a mechanism for consuming an SCA service over the default binding from a non-SCA component.

1. Determine if you are developing the SCA service client starting with an existing WSDL file or with an existing Java interface.
2. Develop the client Java interfaces and data types from a WSDL file if you are not starting with an existing Java interface. Use the wsimport command to generate the SCA service client Java interfaces.
3. Create the reference proxy based on the Java interface.
   a. Create a reference proxy field or setter method that has the same type as the generated Java interface
   b. Annotate this field or setter with the @Reference annotation.

   Now you have completed the steps required to add the reference to your Java component implementation
4. Create a component definition using the Java implementation.

   In the composite definition, add a `<reference>` element that refers back to the original interface and the field or setter of your SCA implementation. The reference is added as a child element of your component. The component is part of a composite definition.

   The `<reference>` name attribute must correspond to the field or setter that contains the @Reference annotation. For a field that contains the @Reference annotation, the name attribute must match exactly. For a setter that contains the @Reference annotation, use the usual Java conventions for translating an annotated setter into a corresponding field, which in turn must match the name attribute.

   For the interface:

   - If your SCA client development started with an existing WSDL file, create an `<interface.wsdl>` element as a child element of the `<reference>` element that points to the WSDL portType.

- If your SCA client development started from existing Java interface, create an <interface.java> element as a child element of the <reference> element that points to the original Java interface. This is optional, since the runtime environment can introspect the Java interface.

In addition to these aspects of your component definition described by these development procedures, there are other aspects of defining a component. These aspects include adding bindings, configuring property values, defining intents, attaching policy sets, and resolving references. You can create multiple components using this same implementation, but all component definitions are the same with respect to the <implementation.wsdl> element and <reference> element described in this step.

5. Deploy the SCA component by creating the SCA business level application from a deployable composite.

   In the previous step, you defined a component providing your SCA service within a composite definition. This composite is either a deployable composite, or one that is used recursively as a composite implementation of a component in a higher-level composite. To learn how to deploy the SCA service, read about deploying and administering business-level applications.

Results

You have created an SCA component that can consume an existing SCA service using a WSDL or Java interface.

Example

The following example illustrates using an existing WSDL interface to generate a Java interface that is used to create a Java implementation that is an SCA client. If you are starting with an existing Java interface, begin with step 4 to follow this example.

1. Copy the following sample account.wsdl WSDL file to a temporary directory.

```
<?xml version="1.0" encoding="UTF-8"?>

<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
     xmlns:account="http://www.myaccount.com/account"
     targetNamespace="http://www.myaccount.com/account"
     name="AccountService">

      <wsdl:types>
         <schema targetNamespace="http://www.myaccount.com/account"
             xmlns="http://www.w3.org/2001/XMLSchema"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:account="http://www.myaccount.com/account">

               <element name="computeAccountAverage">
                 <complexType>
                     <sequence>
                         <element name="account" type="account:Account" />
                         <element name="days" type="xsd:int" />
                     </sequence>
                 </complexType>
             </element>
             <element name="computeAccountAverageResponse">
                 <complexType>
                     <sequence>
                         <element name="return" type="xsd:float" />
                     </sequence>
                 </complexType>
```

```
            </element>

            <complexType name="Account">
                <attribute name="accountNumber" type="xsd:int" />
                <attribute name="accountID" type="xsd:string" />
                <attribute name="accountType" type="xsd:string" />
                <attribute name="balance" type="xsd:float" />
            </complexType>

        </schema>
    </wsdl:types>

    <wsdl:message name="computeAccountAverageRequest">
        <wsdl:part element="account:computeAccountAverage"
            name="parameters" />
    </wsdl:message>

    <wsdl:message name="computeAccountAverageResponse">
        <wsdl:part element="account:computeAccountAverageResponse"
            name="parameters" />
    </wsdl:message>

    <wsdl:portType name="AccountService">
        <wsdl:operation name="computeAccountAverage">
            <wsdl:input message="account:computeAccountAverageRequest" name="accountReq"/>
            <wsdl:output message="account:computeAccountAverageResponse" name="accountResp"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="AccountServiceSOAP" type="account:AccountService">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="computeAccountAverage">
           <soap:operation
                soapAction="computeAccountAverage" />
            <wsdl:input name="accountReq">
               <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output name="accountResp">
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>

     </wsdl:binding>

    <wsdl:service name="AccountWSDLService">
        <wsdl:port binding="account:AccountServiceSOAP"
            name="AccountServicePort">
            <soap:address location=""/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

2. Run the wsimport command from the *app_server_root*\bin\ directory.

   app_server_root\bin\wsimport.bat -keep -verbose account.wsdl

   Run the wsimport command,

   app_server_root/bin/wsimport.sh -keep -verbose account.wsdl

   app_server_root/bin/wsimport -keep -verbose account.wsdl

   After generating the template files using the wsimport command, the following Java files are generated:

```
com/myaccount/account/Account.java
com/myaccount/account/AccountService.java
com/myaccount/account/AccountWSDLService.java
com/myaccount/account/ComputeAccountAverage.java
com/myaccount/account/ComputeAccountAverageResponse.java
com/myaccount/account/ObjectFactory.java
com/myaccount/account/package-info.java
```

3. Identify the generated Java interface from the generated classes.

```
//
// Generated By:JAX-WS RI IBM 2.1.1 in JDK 6 (JAXB RI IBM JAXB 2.1.3 in JDK 1.6)
//
package com.myaccount.account;
...
@WebService(name = "AccountService", targetNamespace = "http://www.myaccount.com/account")
...
public interface AccountService {
    /**
     *
     * @param days
     * @param account
     * @return
     *     returns float
     */
    @WebMethod(action = "computeAccountAverage")
    @WebResult(targetNamespace = "")
    @RequestWrapper(localName = "computeAccountAverage", targetNamespace = "http://www.myaccount.com/account",
         className = "com.myaccount.account.ComputeAccountAverage")
    @ResponseWrapper(localName = "computeAccountAverageResponse", targetNamespace = "http://www.myaccount.com/account",
         className = "com.myaccount.account.ComputeAccountAverageResponse")
    public float computeAccountAverage(
        @WebParam(name = "account", targetNamespace = "")
        Account account,
        @WebParam(name = "days", targetNamespace = "")
        int days);

}
```

This code example is a Java interface, not merely a Java class. The @WebService annotation is present in this Java interface. It is important to know that this example is not the same as the generated @WebServiceClient class, com.myaccount.account.AccountWSDLService. This class is not an interface and is actually not needed in your SCA application.

4. Now that you have Java interface either by generating the Java interface from a WSDL file or you have an existing Java interface, you are ready to develop your SCA client from the Java interface.
5. Place the @Reference annotation on a public or protected field or setter, with the same type as your Java interface.

```
package com.myaccount.client;

import bank.process.BankProcess;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.Service;

import com.myaccount.account.*;

@Service(BankProcess.class)
public class AccountClientComponent implements BankProcess {

   // Note the type, 'AccountService', is the Java interface generated from
   // from the WSDL portType
    private AccountService accountServiceRef;

    //
    // Injected by the SCA container
    //
    @Reference
    public void setAccountServiceRef(AccountService accountServiceRef) {
        this.accountServiceRef = accountServiceRef;
```

```
        }

        public String someMethod(String input) {

            //... some business logic ...

            //  We'll show a simple example of JAXB API usage
            ObjectFactory factory = new ObjectFactory();
            Account account = factory.createAccount();
            account.setAccountNumber(4);
            account.setAccountID("CHECKING");

            int days = 5;

            float avg = accountServiceRef.computeAccountAverage(account, days);

            //... the rest of the business logic ...
        }
    }
```

6. Create a component using the component implementation. When using a WSDL portType interface, you must create component definitions in the composite definition that references the original portType along with the SCA Java implementation. In SCA, a component is a configured instance of a component implementation. There are other aspects of defining a component that are not shown here such as configuring bindings, configuring property values, defining intents, attaching policy sets, and resolving references. Shown here are the aspects of component creation that are common for all component definitions using the implementation developed in this example. This example also includes bindings that you can modify or omit for other components using this component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://bank.process/customer"
         name="bpComposite">

      <component name="BankProcessComponent">

          <implementation.java class="com.myaccount.client.AccountClientComponent"/>
         <!-- The @name attribute corresponds to the setter that is annotated with the @Reference annotation. -->

         <reference name="accountServiceRef">
              <!-- This statement specifies the QName of the WSDL portType,
                    "http://www.myaccount.com/account#AccountService" in the syntax as
                   illustrated in the interface.wsdl statement.  -->

             <interface.wsdl interface="http://www.myaccount.com/account#wsdl.interface(AccountService)" />
             <binding.ws uri="http://localhost:9080/BankingComponent/AccountService"/>

  <!-- This example uses the SCA Web services binding.  However, it does not matter which specific binding
 you choose.  You can also choose to use the SCA default binding or the SCA EJB binding.  -->

          </reference>
     </component>
</composite>
```

7. Configure the composite definition when starting with a Java interface.

   The following snippet is another example of the syntax if you develop an SCA client starting with a Java interface rather than with a WSDL portType. To simplify this example, use the same AccountService Java interface from the previous step but in this case, assume that it was not generated from a WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>

<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
     targetNamespace="http://bank.process/customer"
         name="bpComposite">
```

```
    <component name="BankProcessComponent">

       <implementation.java class="com.myaccount.client.AccountClientComponent"/>

       <!-- The @name value corresponds to the setter that is annotated with the @Reference annotation. -->
              <reference name="accountServiceRef">

            <!-- Because the runtime can introspect the interface, it is unnecessary to specify
                 the interface.java in the composite definition. This is what the interface looks like if you include it.

                       <interface.java interface="com.myaccount.account.AccountService"/>  -->

             <!-- The SCA binding type is omitted. It does not matter which specific SCA binding you choose. -->
       </reference>
    </component>
</composite>
```

8. After a component is defined as part of a deployable composite, either directly or recursively through use of one or more layers of components with composite implementation, you are now ready to deploy the SCA component by creating a SCA business level application.

Using business exceptions with SCA interfaces

You can implement exceptions for remotable interfaces in the Service Component Architecture (SCA) environment to provide additional flow of control for error conditions to meet the needs of your business application.

About this task

To develop SCA service implementations, you can use either a top-down development approach starting with an existing Web Services Description Language (WSDL) file or you can use a bottom-up development approach starting from an existing Java interface or implementation. When using either the top-down or bottom-up development methodologies, you can use tools to map business exceptions on remotable interfaces.

In order to achieve the SOA goal of providing an interoperable platform that is both language and technology neutral, the SCA runtime environment takes an XML-centric view of interfaces and data. When working with Java code, the Java API for XML-Based Web Services (JAX-WS) standard is used to define the mapping between Java code and the XML-based Web Services Description Language (WSDL) file. This mapping also includes the Java programming model with respect to exceptions. Exceptions for remotable interfaces in the SCA environment is defined by the JAX-WS specification. This topic describes the best practices for using business exceptions with SCA interfaces.

Differences between business exceptions and fault beans

To better understand the implications of implementing business exceptions in an SCA environment, it is helpful to understand differences between an exception and a fault bean.

The JAX-WS specification distinguishes between a checked exception and the fault bean that it wrappers. However, this distinction might not be clear because a single class can serve the checked exception and the fault bean functions, especially when you use the bottom-up approach of developing an SCA service starting with a Java interface. When you use the top-down development approach of developing an SCA service starting with a WSDL file, section 2.5 of the JAX-WS specification describes the wrapper pattern for how the fault message maps to a Java checked exception that wrappers a fault bean. The fault bean maps to the fault

element and in SCA environments, the mapping is defined by Java Architecture for XML Binding (JAXB) data binding. The fault bean represents the cross-platform view of the fault message data and includes a schema description. You can use the Java exception within the Java runtime environment and as part of the Java programming model. However, the exception is not part of the interoperable data representation.

When developing SCA services using the bottom-up approach, the distinction between an exception, the fault bean, and the mapping from Java to WSDL or XSD schema is clear if you follow the wrapper pattern described in section 2.5 of the JAX-WS specification. If you have existing Java exceptions, use the standard mapping defined in section 3.7 of the JAX-WS specification for service specific exceptions. In SCA environments, these service specific exceptions are referred to as business exceptions. The mapping for the business exceptions is different than the mapping described in section 2.5 of the JAX-WS specification. Because this wrapper pattern only applies for certain exceptions, this approach has limitations when using the bottom-up development approach. The possible limitations of using the wrapper pattern to implement error handling when using bottom-up development of SCA applications provides additional reasons to consider the advantages of the best practice of top-down development of SCA applications.

Top-down development of SCA services implementing a WSDL fault method

It is a best practice to use the top-down methodology to develop SCA service implementations because this approach leverages the capabilities of the XML interface description and provides a greater ease in interoperability across platforms, bindings, and programming languages. To implement a WSDL fault method, you must obtain the WSDL portType element and define a fault message in terms of a fault element. You can then use the wsimport command-line tool to generate the Java code. This tool generates Java exception code that wraps a fault element in the format specified by the Java API for XML-Based Web Services (JAX-WS) specification, section 2.5.

Bottom-up development of SCA services implementing a Java interface or implementation

Bottom-up development of SCA services occurs when you start with existing Java code. Using this development approach, do not design a remotable interface that might cause a technology exception such as java.sql.SQLException. This exception is more appropriate for a local interface rather than a coarse-grained remotable interface.

1. For top-down development of SCA applications, implement a wrapper pattern for business exceptions.

   The wrapper pattern is based on section 2.5 of the JAX-WS specification.

   a. Obtain your WSDL file; for example:

```
<wsdl:types>
    ...
    <element name="errorCode" type="xsd:int"/>
    ...
</wsdl:types>

<wsdl:message name="BadInputMsg">
    <wsdl:part element="tns:errorCode" name="parameters"/>
</wsdl:message>

<wsdl:portType name="GuessAndGreet">
```

```
<wsdl:operation name="sendGuessAndName">
        <wsdl:input.../>
        <wsdl:fault message="tns:BadInputMsg" name="BadInputMsg"/>
```

b. Generate the Java artifacts using the wsimport tool. You can define the fault according to section 2.5 of the JAX-WS specification; for example:

```
Interface
   public Person sendGuessAndName(...) throws BadInputMsg;
```

c. Use the exception wrappering fault; for example:

```
import javax.xml.ws.WebFault;

@WebFault(name = "errorCode", targetNamespace = "....")
public class BadInputMsg extends Exception
{
    private int faultInfo;

    public BadInputMsg(String message, int faultInfo) {
        super(message);
        this.faultInfo = faultInfo;
    }

    public BadInputMsg(String message, int faultInfo, Throwable cause) {
        super(message, cause);
        this.faultInfo = faultInfo;
    }

    public int getFaultInfo() {
        return faultInfo;
    }
}
```

2. For bottom-up development of SCA applications, implement or convert the exception to follow the wrapper pattern or use the default mapping for of a JAX-WS service specific exception.

   If you have a Java business exception, the complexity of this scenario increases, especially if your exception wraps fault data. For example, the exception wraps data such as an error code or an object that it needs to provide to the client that receives the exception. In this scenario, there are two options:

   - Convert the Java business exception to follow the wrapper pattern as described in section 2.5 of the JAX-WS specification.

     Using the wrapper pattern for the exception enables the exception to map easily from the WSDL to Java code format and then from the Java code to WSDL format. If you modify the exception to follow wrapper pattern, you can use the wsgen tool to convert from Java code to WSDL and later use the wsimport tool to convert from WSDL to Java code, the exception is similar to the one that you modified. To achieve this end goal, you must perform the following steps:

     a. Add constructors that take the fault bean as input parameters.
     b. Implement a public getFaultInfo() method that returns the fault bean.
     c. Add the @javax.xml.ws.WebFault annotation. See the exception wrappering fault example.

   - Use the default mapping of a JAX-WS service specific exception or business exception as described in section 3.7 of the JAX-WS specification.

     If you use the wsgen command-line tool to generate the WSDL, the tool uses this pattern for business exceptions. If you do not generate the WSDL file before deployment, the application server runtime environment implicitly generates the business exception using this pattern.

     Use this option when you:

– cannot change the exception class to follow the JAX-WS wrapper pattern.
– rely on the runtime environment to map the Java code into WSDL such as declaring a `<binding.ws>` binding on a service that is deployed without a WSDL file.

Either of these options work without any additional complexity as long as the exception does not contain fault data.

For exceptions with fault data, the data is handled correctly for each field that contains a public getter or setter method. However, data is lost without a getter or setter pair. In other words, we will serialize or deserialize the exception by viewing it as a Java code.

When using this second option, the following items are important:

– The supported fault pattern is not easily determined. One exception with fault data and also with the getter and setter methods is that some are handled correctly while others are not. Running the wsgen tool at development time generates the schema based on the exception getter methods without assuring that the corresponding setter methods exist in order to populate the exception during unmarshalling.
– If you run wsimport tool against the generated WSDL, you get a different exception class. Your client and service programming model are different which might confuse the Java programmer. However, this generated Java exception follows the pattern described in the JAX-WS specification in section 2.5. You might need to add customization for JAXB data binding in order to generate the client. The results can produce exception names similar to `MyException_Exception`.
– Although the service-specific exception pattern is described in section 3.7 of the JAX-WS specification, not all details for the pattern are specified. Other software implementing JAX-WS might implement the pattern differently. This is not critical, since the WSDL file is interoperable across platforms.

Example

The following examples illustrates using the bottom-up development of SCA applications and using the business exception mapping as described in section 3.7 of the JAX-WS specification.

Example 1: No fault

The string message is the fault in this example, and it is serialized and deserialized successfully.

```
public class RealSimpleException extends Exception {
    public RealSimpleException(String message) {
        super(message);
    }
    public RealSimpleException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Example 2: Exception as JavaBeans

This example works correctly because the string `userdata` fault has associated public getter and setter methods. The string message is also handled correctly.

```
public class TestException extends Exception {

    private String userdata;

    public TestException(String message) {
        super(message);
    }

    public TestException(String message, String userdata) {
        super(message);
        this.userdata = userdata;
    }

    public String getUserdata() {
        return userdata;
    }

    public void setUserdata(String userdata) {
        this.userdata = userdata;
    }

}
```

Example 3: Exception does not follow pattern

This example does not work correctly because the errorCode fault data does not have a setter method. The SCA runtime is not able to correctly determine how to populate the exception with this fault data. The exception occurs, but it is displayed with data loss.

```
package java.sql;

public class SQLException extends Exception ... {
 ...
 public SQLException(String theReason, String theSQLState, int theErrorCode) ...

 public int getErrorCode()
}
```

Considerations for developing SCA applications using EJB bindings

When developing Service Component Architecture (SCA) applications that you intend to use with Enterprise JavaBeans (EJB) bindings, keep in mind that the SCA EJB binding is architected in a Java-centric manner, in contrast to the XML-centric implementations of the SCA default binding and the SCA Web services binding.

The EJB transports marshal and unmarshal application data into the wire format by using Java serialization, whereas the Web services and default bindings use XML serialization. This difference also affects the programming model in that the SCA clients and implementations using the EJB binding must use java.io.Serializable types, in contrast to the Java Architecture for XML Binding (JAXB) data types-based programming model that is used for the SCA default and Web services bindings.

SCA reference

In this case, you have an existing EJB that you want to invoke with an SCA client using a reference that is configured with an EJB binding.

When you develop an SCA client that will invoke an existing EJB using the SCA EJB binding, you must use a Java interface when developing the SCA client rather

than using a WSDL interface. The EJB binding marshalling of application data into the wire data format is performed using Java serialization, not XML serialization as defined by JAXB.

To learn more about SCA references, read about developing SCA service clients. However, when you are using the SCA EJB binding, the information in this topic takes precedence.

Because you obtain the Java interface and parameter types from the EJB provider for use in your client, you do not have to worry about the effects of marshalling and unmarshalling when writing your client. However, when you provide these data types across new services, problems can occur if you pass these data types across new services, because they might not serialize correctly over other bindings, such as the default binding, because of the difference in Java serialization and the JAXB XML marshalling and unmarshalling.

The following example illustrates the problematic scenario of starting with an existing EJB interface and using a Java serializable data type that does not serialize well using JAXB marshalling and unmarshalling.

```
public interface NameService extends javax.ejb.EJBObject {

    public String computeName(Person p) throws RemoteException;
}

// This snippet is intended as an example of a type that is problematic.
public class Person implements java.io.Serializable {

    private int code;
    private String name;

    // The code field must be passed into constructor.  However, this causes problems for
    // for SCA default and Web Services bindings that use JAXB marshalling/unmarshalling.

    public Person(int code) {
        this.code = code;
    }

    public Person() {
    }

    public String getName() {
        return name;
    }

    public void setName(String value) {
        this.name = value;
    }

}
```

The following SCA client A example works correctly. The Person object that is instantiated directly in the ClientAImpl implementation is correctly marshalled to invoke the EJB with a remote interface of NameService.

```
// Client A  ClientAImpl.java
 .....
@Reference
public NameService nameService

public someClientMethod() {
```

```
    // No problem when Person object is instantiated by client
    Person person = new Person(5);
    String name = nameService.computeName(person);
}
```

In contrast, the following example demonstrates the problem with the Person type. The client code has been refactored so that it contains a reference to NameService, and it obtains the Person object that is passed into the computeName method over a new remotable interface, rather than constructing it directly.

```
// Problem client interface

import org.osoa.sca.annotations.Remotable;
@Remotable
public interface PersonFilter {
     boolean filterPerson(Person p);
}


// Problem client implementation

@Service(PersonFilter.class)
public class PersonFilterImpl implements PersonFilter {
     @Reference
     public NameService nameService

     boolean filterPerson(Person p) {
     // ...  business logic
    String name = nameService.computeName(person);
     // ...  business logic
     }
}
```

If the PersonFilterImpl class receives a Person object from the client over the PersonFilter interface and the implementation is invoked using the SCA default binding, the data is not handled correctly. The default binding does not preserve the code field of the Person object that is passed to the PersonFilterImpl class.

For a class without JAXB annotations, JAXB marshalling and unmarshalling preserves JavaBeans properties, but not private data such as the code field, which does not have a setter and is only established in the constructor. When the Person object is passed to the NameService EJB, the code value is set to the default value of 0 regardless of what the PersonFilter client passed to the PersonFilterImpl class.

If the Person type was written in the JavaBeans style with getters and setters for all important data, then this type works correctly in the example for the PersonFilterImpl client. However, if you are consuming an existing EJB, you do not have control over the types it already uses on its interface. Not all existing Java types are optimal for SCA Java programming. To address the problems in this example, you must create a new type for use on the PersonFilter interface and translate the data for this type into a Person object within the PersonFilterImpl class which directly invokes the EJB with the remote interface NameService.

In this example, if the PersonFilter interface was defined as a local interface, then the concerns with preserving data integrity do not apply. The runtime environment performs pass-by-reference semantics across local interfaces that are appropriate for tightly-coupled clients and services such that no data is copied.

SCA service

If you write a new SCA service and intend to expose it over the SCA EJB binding so that an EJB client can invoke the service, it is a best practice to develop the SCA service using the top-down methodology starting with an existing WSDL file or XSD schema and generating the JAXB classes that are used to write the service implementation. Using this approach, you can easily address the differences between Java serialization and JAXB marshalling and unmarshalling by specifying that the generated JAXB classes are Java serializable.

To enable the generated JAXB classes to work correctly over the SCA EJB binding, add the serializable customization to the schema definition so that the generated JAXB classes are Java serializable and implement the java.io.Serializable interface. For example:

```
<schema targetNamespace="http://com.mycompany/banking/" jaxb:version="2.0"
 xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
 xmlns="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <appinfo>
      <jaxb:globalBindings>
        <jaxb:serializable uid="....."/>
     </jaxb:globalBindings>
    </appinfo>
  </annotation>

   <!-- Continue with the rest of the schema definition-->

</schema>
```

As a result, you can use your Java implementation with the generated JAXB data types over the EJB binding, which uses Java serialization. At this point, because you have the generated JAXB artifacts, you can also use your Java implementation with the generated JAXB data types over the SCA default and SCA Web Service bindings, which use XML serialization as defined by JAXB.

If you develop your SCA service using the bottom-up approach starting with Java code, you must use types that implement the java.io.Serializable interface as required when writing an EJB. See the developing SCA services with existing Java code documentation for more information regarding requirements for your user-defined types. Also, see the SCA reference section to learn how to avoid problems with your user-defined types when using EJB bindings because of the differences between Java serialization and JAXB marshalling and unmarshalling.

Chapter 6. Specifying bindings in an SCA environment

After you develop an Service Component Architecture (SCA) component, you can use bindings to specify how SCA services and references enable the component to communicate with other applications.

About this task

Services and references enable a component to communicate with other applications. By design, however, the SCA services and references say nothing about how this communication occurs. Bindings are used to determine how a component communicates with the world outside its domain. SCA services use bindings to describe the access mechanism that clients must use to call the service. SCA references use bindings to describe the access mechanism that is used to call a service. Depending on what the SCA component is communicating with, a component might or might not have explicitly specified bindings.

The Feature Pack for SCA supports the following binding types:

- SCA binding

  The SCA binding is also referred to as the default binding. It is the binding that is used when no other binding is specified for configuration of a component reference or service. It is the natural binding to use when your SCA client invokes an SCA service in the same domain. It is not intended to be interoperable in any way with other SCA runtime implementations. Components communicating within the same domain only need to explicitly configure a default binding on a service or reference when there is at least one non-default binding, such as the SCA Web service binding or the SCA EJB binding, that is also configured.
- Web service binding

  The SCA Web service binding applies to the services and references of components. The Web service binding is designed for SOAP-based Web Services-Interoperability (WS-I) compliant Web services. This binding defines the manner in which a service is made available as a Web service, and in which a reference can invoke or access a Web service. The Web service binding enables SCA applications to expose SCA services as Web services to external clients that might or might not be implemented as an SCA component. This binding is a Web Services Description Language (WSDL)-based binding which means that the Web service binding either references an existing WSDL binding or enables you to specify enough information to generate a WSDL file. When an existing WSDL binding is not referenced, you can generate a WSDL binding. You can further customize a SCA Web service binding using SCA policy sets.

  Web services technology plays an important role in most SOA solutions relevant today, including SCA. The SCA Web service binding type enables SCA applications to expose services as Web services to external clients as well as enabling SCA components access to external Web services. External clients that access SCA services exposed as Web services may or may not be implemented as an SCA component. You can use the Web service binding element `<binding.ws>` within either a component service or a component reference definition. When the Web service binding is used with a component service, this binding type enables clients to access a service offered by a particular component as a Web service. When the Web service binding is used with a component reference,

components in an SCA component can consume an external Web service and access as if it was any other SCA component. Only WSDL Version 1.1 is supported.

- EJB binding

  EJB session beans are a common technology used to implement business services. The ability to integrate SCA with services based on session beans is useful because it preserves the investment incurred during the creation of those business services, while enabling the enterprise to embrace the newer SCA technology in incremental steps. The simplest form of integration is to simply enable SCA components to invoke session beans as SCA services. There is also a need to expose services such that they are consumable by programmers skilled in the EJB programming model. This enables existing session bean assets to be enhanced to exploit newly deployed SCA services without the EJB programmers having to learn a new programming model.

  The SCA EJB binding enables SCA to integrate with existing Java EE applications. It exposes SCA services as stateless session beans to external clients. The binding element `<binding.ejb>` is used within a component service or component reference definition. Support is provided for the EJB binding when using both 2.x and 3.0 EJB styles for both the SCA service and reference.

1. Select a binding type to use for an SCA component.
   - Use the SCA default binding when you want to invoke an SCA service from an SCA client.
   - Use the SCA Web service binding to specify that an SCA service is made available as a Web service or an SCA reference can invoke a Web service.
   - Use the SCA EJB binding to integrate SCA with services based on session beans.
2. Configure the selected binding and use it in an SCA component or application.

Results

SCA components can use the configured bindings to communicate with other SCA services and references.

What to do next

Deploy the SCA component or application.

Configuring the SCA default binding

You can configure the Service Component Architecture (SCA) default binding for services and references.

About this task

Bindings determine how a component communicates with the world outside its domain. Services use bindings to describe the access mechanism that clients must use to call the service. References use bindings to describe the access mechanism used to call a service. The SCA binding is also referred to as the default binding. The default binding is the binding that is used when no other binding is specified for a configuration of a component reference or service. Use this binding when an SCA client invokes an SCA service in the same domain. It is not intended to be interoperable in any way with other implementations of SCA runtime environments.

- Configure an SCA service with the SCA default binding.

If the service is only exposed over the default binding, then you do not need to explicitly add the <binding.sca> element because this binding is default binding for SCA. If your SCA service has more than one binding and the SCA default binding must be one of them, you must specify the <binding.sca> element in the composite definition.

- Configure an SCA reference with the SCA default binding.

  For the reference, you also do not need to specify the <binding.sca> element. For an reference with a default binding, the reference specifies a target attribute indicating the target service. To indicate the target at the reference, specify target=componentName/serviceName. If only one service exists for the service component, then you only need to specify the componentName; for example: target=ComponentName.

Results

You have implicitly or explicitly configured the SCA default binding for your SCA service or reference.

Example

The following examples illustrate multiple scenarios for configuring the SCA default bindings.

Top level composite with SCA service binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples" name="MyComposite">

      <component name="HelloWorldServiceComponent">
         <implementation.java class="test.HelloWorldImpl"/>
      </component>

   </composite>
```

Top level composite with SCA reference binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
       name="ClientComposite">


      <component name="ClientComponent">
        <implementation.java class="test.GreetingsServiceImpl"/>
        <reference name="helloWorldService" target="TargetComponent"/>
      </component>

   </composite>
```

OR:

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
        name="ClientComposite">


      <component name="ClientComponent">
        <implementation.java class="test.GreetingsServiceImpl"/>
        <reference name="helloWorldService" target="TargetComponent/HelloWorld"/>
        <!-- compName/serviceName -->
        </component>

   </composite>
```

Top level composite with SCA service binding with transaction policy attribute defined

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://neworder/sca/jdbc"
       name="NewOrderComposite">


      <component name="NewOrderServiceComponent">
         <service name="NewOrderService" requires="propagatesTransaction.false"/>
         <implementation.java class="neworder.sca.jdbc.NewOrderServiceImpl" requires="managedTransaction.local"/>
      </component>

   </composite>
```

Top level composite with SCA service binding supporting WSDL interface

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  targetNamespace="http://mysca/samples"
         name="ClientComposite">

        <component name="ClientComponent">
           <service name="HelloWorldService">
             <interface.wsdl interface="http://helloworld#wsdl.interface(HelloWorld)"/>
           </service>
          <implementation.java class="test.HelloWorldImpl"/>
         </component>

   </composite>
```

Top level composite with SCA reference binding supporting WSDL interface

```
<?xml version="1.0" encoding="UTF-8"?>

    <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples" name="ClientComposite">


       <component name="ClientComponent">
         <implementation.java class="test.GreetingsServiceImpl"/>
         <reference name="helloWorldService" target="MyServiceComponent">
            <interface.wsdl interface="http://helloworld#wsdl.interface(HelloWorld)"/>
         </reference>
       </component>

</composite>
```

Intra composite over SCA default binding

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://mysca/samples"
         name="Service1Composite">

       <component name="HWServiceComponent">
         <implementation.java class="test.HelloWorldImpl"/>
         <reference name="component2Ref" target="Component2"/>
      </component>


        <component name="Component2">
        <implementation.java class="test.Component2Impl"/>
        </component>

   </composite>
```

SCA Service with workManager specified for the service

```
<?xml version="1.0" encoding="UTF-8"?>

   <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"  xmlns:wm="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
         targetNamespace="http://mysca/samples" name="Composite2">


      <component name="Component2">
        <service name="OneWayService">
```

```
      <!-- This service uses the @oneway annotation to specify this operation only has an input message
            and no output message.   -->
        <wm:workManager value="wm/scatest"/>
        <!-- This service specifies a workManager where the jndiName is wm/scatest. -->
    </service>
    <implementation.java class="test.Component2Impl"/>
    <reference name="component3" target="Component3"/>
  </component>


  <!-- component service with @oneway (non blocking operation -->
  <component name="Component3">
    <!-- By not defining the workManager, the SCADefaultWorkmanager that is created by the SCA
         runtime environment is used here. -->
    <implementation.java class="test.Component3Impl"/>
  </component>

</composite>
```

Using the SCA default binding to find and locate SCA services

Support exists for an API that is specific to WebSphere Application Server that you can use to find and invoke Service Component Architecture (SCA) services over the SCA default binding.

About this task

According to the SCA Version 1.0 specification, you can only obtain a reference to an SCA service from another component that is statically wired to the service. However, in the Feature Pack for SCA, you can use a service proxy to invoke the target service. This function requires a WebSphere Application Server base or network deployment topology with at least one server that has the Feature Pack for SCA installed. Also, SCA service must be deployed, running, and accessible over the default binding, `<binding.sca>`. There is no support for a domain URI, so all requests go to the default domain at the cell level. Using this API enables code that is not an SCA component to use the SCA client programming model.

Use the import method in your client code to locate an SCA service.
The following method is supported for client code to locate a service that is deployed onto a cell:

```
import com.ibm.websphere.sca.context.CurrentCompositeContext;
import com.ibm.websphere.sca.context.CompositeContext;
CompositeContext compositeContext = CurrentCompositeContext.getContext();
EchoService echoService = (EchoService) compositeContext
 .getService(EchoService.class, "SCASimpleEchoService");
```

Configuring the SCA Web service binding

You can expose a Service Component Architecture (SCA) application as a Web service by configuring the SCA Web service binding.

About this task

Web services technology plays an important role in most service-oriented architecture (SOA) solutions relevant today, including SCA. The Web service binding type enables SCA applications to expose services as Web services to external clients as well as enabling SCA components access to external Web services. External clients that access SCA services exposed as Web Services might or might not be implemented as an SCA component. You can use the Web service binding element `<binding.ws>` within either a component service or a component reference definition. When this binding is used with a component service, the Web

service binding type enables clients to access a service that is offered by a particular component as a Web service. In the case where the Web service binding is used with a component reference, components in an SCA composite can consume an external Web service and access it just like any other SCA component. The Web service binding supports Web Services Description Language (WSDL) Version 1.1.

The SCA Web service binding applies to the services and references of components. The Web service binding defines the manner in which a service is made available as a Web service, and in which a reference can invoke a Web service. This binding is a WSDL-based binding; meaning that the binding either references an existing WSDL document or enables you to specify enough information to generate a WSDL document.

Note: The SCA Web service binding provides support for providing and consuming services using the SOAP Version 1.1 over HTTP and SOAP V1.2 over HTTP protocols.

Note: The Feature Pack for Service Component Architecture (SCA) does not support the following functions:

- Java API for XML-Based Web Services (JAX-WS) handlers when using the SCA Web service binding
- Message Transmission Optimization Mechanism (MTOM) or SOAP with Attachments (SwA) binary message optimizations

Use the SCA Web service binding without implementing JAX-WS handlers. Do not use SwA binary message optimizations or MTOM optimizations for transferring binary data between SCA clients and services that use the SCA Web service binding. Instead of implementing MTOM or SwA binary message optimizations to send binary data, use the base64Binary XML Schema Definition (XSD) encoding to embed the data within the SOAP message.

- Configure an SCA service with the SCA Web service binding.

  Depending on whether you develop your SCA service using the top-down development approach starting with an existing WSDL file or you develop your SCA service using the bottom-up development approach starting with existing Java code, you might or might not have a WSDL file available. Additionally, the WSDL file might only define a portType or it might include a port definition as well.

  – For SCA applications that are developed top-down starting from a WSDL port, you must refer to the port definition in the existing WSDL file by adding a `<binding.ws>` element as a child of your `<service>` element. The following provides an example of the syntax for this step:

```
<binding.ws wsdlElement="<port target Namespace>#wsdl.port(<service name attr>/<port name attr>)"/>
```

The location attribute of the `<address>` element for the port is ignored by the runtime environment in determining the URL at which your service is invoked.

The following example demonstrates the relationship between the WSDL file and the composite definition for this scenario:

WSDL file

```
<wsdl:definitions targetNamespace="http://www.ibm.com/" xmlns:tns="http://www.ibm.com/" ...>
      ....
     <wsdl:portType name="MyPortType ">
      ....
     <wsdl:binding name="MyBinding" type="tns:MyPortType">
```

```
      ....
     <wsdl:service name="MyService">
         <wsdl:port binding="tns:MyBinding" name="MyPort">
             <wsdlsoap:address location=""/>
         </wsdl:port>
     </wsdl:service>
```

Composite definition

```
<composite...>
     <component name="MyComponent">
       <implementation.java class="test.MyCompImpl"/>
         <service name="GuessAndGreetWrapped">
             <interface.wsdl interface="http://www.ibm.com/#wsdl.interface(MyPortType)" />
             <binding.ws wsdlElement="http:// www.ibm.com/#wsdl.port(MyService/MyPort)" />
         </service>
     </component>
      ...
</composite>
```

– For SCA applications that are developed top-down starting from a WSDL portType, you must create an empty <binding.ws> element as the child of your <service> element. Creating the empty <binding.ws> element directs the runtime environment to generate a port that corresponds to your WSDL portType definition. The generated port uses a SOAP 1.1 over HTTP WSDL binding.

  The following example demonstrates the relationship between the WSDL file and the composite definition for this scenario:

  WSDL file

```
<wsdl:definitions targetNamespace="http://www.ibm.com/" xmlns:tns="http://www.ibm.com/" ...>
      ....
     <wsdl:portType name="MyPortType ">
```

Composite definition

```
<composite...>
     <component name="MyComponent">
       <implementation.java class="test.MyCompImpl"/>
         <service name="GuessAndGreetWrapped">
             <interface.wsdl interface="http://www.ibm.com/#wsdl.interface(MyPortType)" />
             <binding.ws/>
         </service>
     </component>
      ...
</composite>
```

– For SCA applications that are developed bottom-up starting from existing Java code, you must create an empty <binding.ws> element as the child of your <service> element. Creating the empty <binding.ws> element directs the runtime environment to generate a WSDL portType that corresponds to your Java interface, and a port with a SOAP 1.1 over HTTP WSDL binding.

  The following example demonstrates the <binding.ws> element within the composite definition for this scenario:

  Composite definition

```
<composite...>
     <component name="MyComponent">
       <implementation.java class="test.MyCompImpl"/>
         <service name="GuessAndGreetWrapped">
             <binding.ws/>
         </service>
     </component>      ...
</composite>
```

• Test the endpoint for your deployed SCA service. After you configure the SCA service with an SCA Web service binding, you can test the endpoint for your deployed SCA service. The URL format for the endpoint is the following: `http://<host>:<port_of_default_host>/Component_name/Service_name`

- Configure an SCA reference (client) with an SCA Web service binding.
  1. Configure the reference with an <interface.wsdl> element that refers to the portType of the target service. After you have obtained a WSDL portType from the Web services provider, you can configure the <interface.wsdl>. Read about developing SCA service clients to learn how to configure the reference with an <interface.wsdl>.
  2. Resolve the SCA reference to an actual endpoint of a deployed Web service using one of the mechanisms provided by the SCA Web service binding support.
     - When the target Web service is deployed as an SCA component service in the same domain as the client component, you can resolve the reference to the target component using the <reference> @target attribute. Using this attribute eliminates the need to know the specific URL of the deployed target service. For example:

```
<!-- The composite definition for the target service. -->
    <component name="TargetComponent">
      <implementation.java .../>
        <service name="MyService">
            <interface.wsdl ... />
            <binding.ws ... />
         </service>
     </component>

<!– The composite definition for the client when the client is in same SCA domain as the target component. -->
    <component name="ClientComponent">
      <implementation.java .../>

        <!-- Resolution done using the @target annotation. -->
        <reference name="myRef" target="TargetComponent/MyService">
             <interface.wsdl ... />

             <!-- The binding does not need endpoint-related info added. -->
             <binding.ws/>
        </reference>
     </component>
```

     - If the target Web service is not an SCA service in the same domain as the client, you must use a binding-specific endpoint resolution mechanism. You can also resolve a reference to an SCA service in the same domain by using the binding-specific mechanisms, instead of using the @target annotation.
       a. You can define the endpoint for a deployed Web service in an existing WSDL file that is obtained from a service provider. In this case, the client composite definition refers to the WSDL port of the target service which also specifies the endpoint from the <wsdlsoap:address> @location attribute. For example:

          ```
          <wsdlsoap:address location="http://host:port/ComponentName/ServiceName"/>
          ```

          The client points to the WSDL port using a @wsdlElement attribute on the <binding.ws> element using the following syntax:

          ```
          <port target Namespace>#wsdl.port(<service name attr>/<port name attr>)"/>
          ```

          The following example WSDL and composite definitions illustrate defining an endpoint for a deployed Web service.

```
<!-- An example WSDL file.   -->

<wsdl:definitions targetNamespace="http://my.work/test/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"   ... >
    <wsdl:service name="MyService">
      <wsdl:port binding="..." name="MyPort">
        <wsdlsoap:address
           location="http://www.mywork.com:9080/TargetComponent/MyService "/>
     </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

```
<!-- An example composite definition.   -->
     <component name="ClientComponent">
      <implementation.java .../>
        <reference name="myRef">
           <interface.wsdl ... />
           <binding.ws wsdlElement="http://my.work/test/#wsdl.port(MyService/MyPort)"/>
        </reference>
     </component>
```

b. You can add the endpoint to the composite definition when the endpoint is not present in the WSDL file. In this case, add the @uri attribute to the <binding.ws> element to specify the endpoint.

```
<!-- An example WSDL file.   -->

<wsdl:definitions targetNamespace="http://my.work/test/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"   ... >
    <wsdl:service name="MyService">
      <wsdl:port binding="..." name="MyPort">

        <!-- Here, the endpoint not specified in the WSDL. -->
         <wsdlsoap:address   location=""/>
     </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

```
<!-- An example composite definition.   -->
     <component name="ClientComponent">
      <implementation.java .../>
        <reference name="myRef">
             <interface.wsdl ... />
             <binding.ws wsdlElement="http://my.work/test/#wsdl.port(MyService/MyPort)"
             uri="http://www.mywork.com:9080/TargetComponent/MyService" />
        </reference>
     </component>
```

- (optional) If your service or reference interface is bidirectional such that a callback is defined, you must also configure the Web service binding on the callback.
  - Configure callback for your SCA Web service binding using the WSDL port.

    Configuring callback for your SCA Web service binding is similar to configuring an SCA service with the SCA Web service binding; however, you add a second service, the callback. When you configure an SCA service with the SCA Web service binding, you define a WSDL port, either explicitly by pointing directly to a WSDL port definition, or implicitly by giving the runtime enough information to calculate a WSDL port. Similarly, you must also define a WSDL port for the callback. A difference for the callback is that the runtime environment defines the WSDL port that is used for the callback because the runtime environment must keep this port tightly coupled to the forward call. Therefore, the most you can do when developing your SCA application using the top-down approach and defining a callback for this service, is to point to a WSDL binding. For example:

```
<!-- Configuring a service with callback with Web service binding -->
      <component name="HelloWorldServiceComponent">
            <implementation.java class="..." />
            <service name="HelloWorldService">
                 <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                      callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
                 <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.port(HelloWorldService/HelloWorldSoapPort)"/>
                 <callback>
                     <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.binding(HelloWorldCallbackSoapBinding)" />
                 </callback>
            </service>
     </component>
```

The following is an example of a configuration of a client component with reference to this service with callback defined. The reference and service configuration each share the same view of which direction is the forward direction and which is the callback direction.

```
<!-- Configuring a reference with callback with Web service binding. -->
      <component name="HelloWorldClientComponent">
            <implementation.java class="..." />
            <reference name="helloWorldRef">
                 <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                  callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
            <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.port(HelloWorldService/HelloWorldSoapPort)"/>
            <callback>
                <binding.ws wsdlElement="http://www.ibm.com/sca/#wsdl.binding(HelloWorldCallbackSoapBinding)" />
            </callback>
          </service>
     </component>
```

– Configure callback for your SCA Web service binding using the WSDL portType.

  Similar to the scenario of configuring a service with a Web service binding when starting with a WSDL port, you also configure an empty `<binding.ws>` element to configure callback using the WSDL portType. The following composite definition example illustrates the scenario when starting with two WSDL portType definitions that has such that one interface uses a forward direction and the other interface uses callback:

```
<!-- Configuring a service with callback with Web service binding -->
      <component name="HelloWorldServiceComponent">
            <implementation.java class="..." />
            <service name="HelloWorldService">
                <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                 callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
                <binding.ws/>
                <callback>
                   <binding.ws/>
                </callback>
            </service>
     </component>

<!-- Configuring a reference with callback with Web service binding -->
      <component name="HelloWorldClientComponent">
            <implementation.java class="..." />
            <reference name="helloWorldRef">
               <interface.wsdl interface="http://www.ibm.com/sca/#wsdl.interface(HelloWorld)"
                callbackInterface=" http://www.ibm.com/sca/#wsdl.interface(HelloWorldCallback)"/>
               <binding.ws/>
               <callback>
                 <binding.ws/>
               </callback>
            </service>
     </component>
```

– Configure callback for your SCA Web service binding using the Java interface.

  For the bottom-up case starting with a Java interface, the composite definition is identical to the WSDL port and portType scenarios, except that you must replace the `<interface.wsdl>` elements with the `<interface.java>` element. For example:

```
<interface.java interface="helloworld.HelloWorldService"
         callbackInterface="helloworld.HelloWorldCallback"/>
```

Results

You have a configured SCA Web service binding service or reference.

Note:

There are additional ways for the Web service binding function to generate a WSDL port that are not described in this topic. However, these additional methods that rely on WSDL generation at run time add dependencies on the runtime environment that can cause problematic results.

For an example that is not problematic, suppose you write a service using the bottom-up style, starting from a Java interface, and deploy the service

with an empty element. This directs the runtime environment to generate the WSDL port for this service. Also suppose an SCA client is developed with access to the original Java classes used to write the service implementation. This SCA client is used to test the SCA service using a client-side reference with Web service binding. You can configure this reference without any knowledge of the service WSDL. In this case, the reference interface is the original Java interface of the service, and you can resolve the reference using the `<reference>` @target mechanism. See the resolving SCA references documentation for more information on using the `<reference>` element to resolve an SCA reference using the @target attribute. Using this approach, there is no WSDL to obtain or refer to in constructing the client. This works because the Feature Pack for SCA runtime environment maps the service-side Java to WSDL in an identical manner as it maps the client-side Java to WSDL.

In contrast, the following scenario is problematic. Suppose that you write an SCA client with a Web service binding reference to a Web service that is hosted on a platform other than the Feature Pack for SCA. It might seem reasonable to generate your Java client from the service provider, and then ignore the WSDL from that point on, avoiding the additional syntax in your client-side composite definition. To do this, you use the `<binding.ws>` element @uri attribute to specify the endpoint URL where the service is hosted. This scenario is problematic because it forces the runtime environment to generate a WSDL port for the client which might result in subtle mismatches between the WSDL generated for the client side and the actual WSDL port description of the deployed Web service.

You can avoid these potential problems by ensuring that client package references the original WSDL obtained from a Web service provider. If you use the shortcut of omitting a client-side reference to the WSDL, be sure to do so only in the case when you are sure the WSDL port that is generated for the client is identical to the WSDL port of the deployed service because the service port is generated using the same algorithm.

Configuring Web service binding custom endpoints to support a proxy server

You can configure SCA composites that are accessed by Hypertext Transfer Protocol (HTTP) for custom service endpoints using Web services bindings.

Before you begin

Before you begin this task, install your Service Component Architecture (SCA) application.

About this task

When a service is exposed over the SCA Web service binding, the service endpoint is specific to the server in which the service is hosted. Clients use this endpoint URI to access the service. In some cases, you may want clients to indirectly reference the service by using a proxy server as the service endpoint. For example, a proxy server is required to implement clustered Web service binding endpoints. To enable clients to use a proxied endpoints, there are two ways to do this:

- If your endpoints are specified in the SCA contributions composite definition or WSDL document location attribute, you must specify the proxy server endpoint instead of the WebSphere server specific endpoint.

- If your client resolves the endpoint by using the <reference target=""> attribute in your client composite definition, use the administrative console to configure the custom endpoints for SCA composites that are accessed by the Hypertext Transfer Protocol (HTTP) protocol. This approach is the most flexible for SCA clients within the same domain as their service providers. When using <reference target=""> attribute, SCA references can resolve the service endpoints without the client specifying endpoints in the composite definition or WSDL document.

1. Open the administrative console.
2. In the navigation pane, expand **Applications** → **Application Types** → **Business-level applications** → *application_name* → *composition_unit_name* → **Provide HTTP endpoint URL information**.
3. Select the HTTP endpoint URL prefix. When entering custom endpoints you must specify one and only one endpoint URL prefix each for the HTTP and HTTPS protocols.

Results

You have configured Web services bindings custom endpoints.

What to do next

You can configure the bindings to do transport layer authentication.

Routing requests to an SCA service exposed over the SCA Web service binding when using external Web servers

If you are using an external Web server to route requests to an SCA service that is exposed over the SCA Web service binding, you must define the endpoints of the SCA service to the Web Server HTTP plug-in.

About this task

Requests to services that are exposed over the SCA Web service binding that use the proxy server type that is provided with WebSphere Application Server are routed over the specified proxy by default.

However, if your configuration uses an external Web server with the HTTP plug-in for WebSphere Application Server and you want requests to services that are exposed over the SCA Web service binding to route through the external Web server, you must define the endpoints for the SCA service by adding the service URL patterns to the plugin-cfg.xml file for the Application Server.

1. Obtain the URL patterns for each service. You can obtain the URL patterns in one of the following ways:
   - Use the message, which is located in the server log file, that indicates the Web application is successfully created.

     During service startup, for each service that is exposed over the SCA Web Service binding, a dynamic Web application is created and configured with the URI of the service. This process is described as an informational message within the server log file as shown in the following example.

     In the following example message, the helloworldws composition unit contains the AsyncTranslatorService service, which is exposed over the SCA Web service binding. This message provides the necessary context root and URL pattern for the service, which you must add to the plugin-cfg.xml file.

```
/AsynchTranslatorComponent/AsynchTranslatorService/*

[[11/18/08 10:10:52:156 EST] 00000070 servlet I com.ibm.ws.webcontainer.servlet.ServletWrapper init
SRVE0242I: [helloworldws]
[/AsynchTranslatorComponent/AsynchTranslatorService]
[SCA_WS_BINDING_IMPL_CLASS_PLACEHOLDER]: Initialization successful.
[11/18/08 10:10:52:156 EST] 00000070 WASAxis2Exten I
WSWS7037I: The /* URL pattern was configured for the SCA_WS_BINDING_IMPL_CLASS_PLACEHOLDER servlet located
in the SCAWSBindSERV_AsynchTranslatorComponent_AsynchTranslatorService.war Web module.
```

- Use the warinfo.props WebSphere configuration repository file.

  For each composition unit that has at least one service or reference exposed over the SCA Web service binding, a single warinfo.props file is generated during deployment. This file contains configuration information for each dynamic Web application that starts during the server startup process.

  The warinfo.props file is located in the *profile_root*\SOAAppSrv01\config\ cells\ cell1\cus\helloworldws\cver\BASE\meta\warinfo.props directory.

  The file contains an entry for each dynamic Web application. For example:

  ```
  #
  #Tue Nov 18 10:10:37 EST 2008
  SCAWSBindSERV_AsynchTranslatorComponent_
  AsynchTranslatorService.war=AsynchTranslatorComponent/
  AsynchTranslatorService\:default_host\:false\:false\:false
  ```

  The value immediately following the `war=` and ending prior to the `\:` is the context root for the Web application. In this example, the context root is **AsynchTranslatorComponent/AsynchTranslatorService**.

2. Add the values for each dynamic Web application to the plugin-cfg.xml file.

   After obtaining all of the entries from each of the services that you want to define to the proxy server, add the values to the plugin-cfg.xml file. It is important that you add the URI to the specific UriGroup that contains a server and hosts the proxied service because multiple UriGroups might exist. If this process is not done correctly, an HTTP 404 message results.

   In the following example, see the **AsynchTranslatorComponent/ AsynchTranslatorService** entry that has been added to the list of URI patterns:

   ```
   <UriGroup Name="default_host_Cluster2_URIs">
   ? <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/*" />
   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsp" />
   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsv" />
   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/*.jsw" />
   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/j_security_check" />
   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/IBM_WS_SYS_RESPONSESERVLET/ibm_security_logout" />

   ?  <Uri AffinityCookie="JSESSIONID"
   AffinityURLIdentifier="jsessionid"
   Name="/AsynchTranslatorComponent/AsynchTranslatorService/*" /> </UriGroup>
   ```

Results

You have configured the endpoints for SCA services to route requests through an external Web server configured with the HTTP plug-in for WebSphere Application Server

Using EJB bindings in SCA applications

Use this task to learn how to use Enterprise JavaBeans (EJB) bindings in SCA applications.

Support is provided for EJB bindings in 2.x and 3.0-style beans, for both service, reference, and reference target.

The following is an example of an composite definition that has a service exposed over an EJB 3.0 binding:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0" targetNamespace="http://neworder/sca/jdbc"
name="NewOrderComposite">

<component name="NewOrderEJB3ServiceComponent">
<implementation.java class="neworder.sca.jdbc.NewOrderServiceImpl" requires="managedTransaction.local"/>
<service name="NewOrderService" requires="suspendsTransaction">
<interface.java interface="neworder.sca.jdbc.NewOrderService"/>
<binding.ejb ejb-version="EJB3"/>
</service>
</component>
</composite>
```

A client that wants to invoke the resultant enterprise bean would treat it like any other enterprise bean and not like a regular SCA service. CompositeContext.getService is not supported for a non-SCA binding, therefore, a getService() on the CompositeContext would not work here. The following is the client code for the above example:

```
InitialContext ctxt = new InitialContext();
Object remoteObj =
  ctxt.lookup("ejb/sca/ejbbinding/NewOrderEJB3ServiceComponent/NewOrderService#neworder.sca.jdbc.NewOrderServiceRemote");
NewOrderServiceRemote newOrderRemote =
  (NewOrderServiceRemote) PortableRemoteObject.narrow(remoteObj, NewOrderServiceRemote.class);
```

The following is an example of an composite definition that contains references to both EJB 2.x and EJB 3.0 bindings:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
targetNamespace="http://erww.workload" name="ConvertComposite">

<component name="ConvertInputOutputServiceComponent">
<implementation.java class="convert.inputoutput.sca.ConvertInputOutputServiceImpl"

<reference name="priceQuoteSessionReference">
<interface.java interface="priceQuoteSession.PriceQuoteSession"/>
<binding.ejb uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#ejb/session/PriceQuoteSessionFacadeBean"/>
</reference>
</component>
</composite>
```

Different binding.ejb attributes can be used for service side EJB bindings or reference side EJB bindings. The following information explains how the default value is calculated for each side:

Service side

The service side EJB binding applies only to Java archive (JAR)-packaged SCA applications.

- EJB 2.0-level beans: URI is the JNDI name for the home; you can define it, or it can be calculated with the default short name in the following form:

/sca/ejbbinding/*<Component_Name>*/*<Service_Name>*

Therefore, the URI can be calculated as:

```
corbaname:iiop:localhost:2812/NameServiceServerRoot#ejb/sca/ejbbinding/<Component_Name>/<Service_Name>
```

You can use it to look up home.

- EJB 3.0-level beans: The URI contains the component-id, therefore, either you can define it, or it is calculated the same as the EJB 2.0 beans as follows:

  sca/ejbbinding/*<Component_Name>*/*<Service_Name>*

  The URI can be calculated as:

```
corbaname:iiop:localhost:2812/NameServiceServerRoot#ejb/sca/ejbbinding/<Component_Name>/<Service_Name>#
  <package.qualified.interface of SCA Java interface with prefix of Remote or Local to the class name
```

  You can use it directly to get the business interface.

  The following code example displays as if it were a session bean:

```
<session name="ServiceNameBean" component-id="sca/ejbbinding/<Component_Name>/<Service_Name>"/>
```

When an SCA service is exposed through an EJB service binding, the service is exposed through an enterprise bean. During deployment, the SCA runtime generates a session bean for the service exposed through the EJB binding. The caller of the composite service can invoke this service by accessing the generated enterprise bean as if they are invoking any enterprise bean.

The generated enterprise bean for the composite service is in the directory, WAS_HOME\AppServer\profiles\PROFILE_NAME\installedApps\CELL_NAME\COMPOSITE_NAME.ear\. Callers need to include the client required classes, such as remote or home, of the generated bean in the classpath or bundle the classes in the JAR file.

Reference side

The reference side EJB binding applies to both JAR-packaged applications and Web archive (WAR)-packaged applications, if not otherwise stated.

- The URI is used to lookup either the EJB 2.x home or EJB 3.0 business interface. Follow the naming convention of the Java Enterprise Edition (JEE) specification if you are using an existing JEE EJB module. If you use an SCA service with the binding.ejb attribute, use the value mentioned above. For more information about the EJB 3.0 JNDI name, see the topic EJB 3.0 bindings overview.
- homeInterface: Not used
- ejb-link-name: Only applies to WAR-packaged SCA applications. When URI is not defined, use it to look up an EJB module that is defined in the same enterprise archive (EAR) as the WAR.
- session-type: default value ″stateless″
- ejb-version: default value ″EJB2″

Attention: A lookup issue for EJB 3.0 reference bindings might occur when the URI follows the corbaname:iiop:host:port/NameServiceServerRoot##<ejb3_binding_longform> pattern This problem exists only for EJB 3.0 reference bindings. When the EJB 3.0 reference binding URI follows the corbaname:iiop:host:port/NameServiceServerRoot##<ejb3_binding_longform> pattern, where *ejb_binding_longform* is ejb/<component-id#<package.qualified.interface>, and if more than one enterprise bean that is implementing the same interface is deployed on that server,

lookup may not be directed to the correct EJB with corresponding component ID.

An example of a URI where this problem can occur is as follows:

```
uri="corbaname:iiop:host:port/NameServiceServerRoot#ejb/EJB3CounterSample/EJB3Beans.jar/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter
```

There are two enterprise beans implementing the com.ibm.websphere.ejb3sample.counter.RemoteCounter interface. To avoid this issue:

- Use a URI that does not start with "corbaname:"
- Use a binding name in the URI that is an EJB binding short form, for example, corbaname:iiop:host:port/NameServiceServerRoot#<package.qualified.interface>.
- Use a binding name in the URI that is a unique user-defined binding name.
- Ensure that the two enterprise beans that are deployed on the server do not implement the same interface.
- Ensure that the EJB binding URI is pointing to an EJB 2.0 bean.

To resolve this problem, follow these guidelines:

- If the EJB reference binding is accessing an enterprise bean that is located in the same cell, the URI should not start with "corbaname:."
- For same cell lookup, the URI pattern should be one of the following.

```
uri="ejb/EJB3CounterSample/EJB3Beans.jar/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
```

or

```
uri="cell/clusters/<cluster_name>/ejb/EJB3CounterSample/EJB3Beans.jar/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
```

or

```
uri="cell/nodes/<node_name>/servers/<server_name>//ejb/EJB3CounterSample/EJB3Beans.jar/StatelessCounterBean#com.ibm.websphere.ejb3sample.counter.RemoteCounter"
```

- Even in cross cell access, the recommended method is to create a namespace binding for the enterprise bean that is accessed by the EJB reference binding. After the namespace binding is created, use the namespace binding in the URI of the EJB reference binding as uri="cell/persistent/<*name_in_namespace_binding*>".

Different patterns of the SCA EJB reference binding URI are based on the user setup and configurations. If the SCA EJB reference binding is accessing a stateless session bean on the same server, the EJB reference binding URI can be designated as the JNDI name, uri="ejb/com/app/resumebank/ResumeBankHome". If the SCA EJB reference binding is referencing another SCA service with an EJB binding in the same server, the URI can be designated as the JNDI name, uri="ejb/com/app/resumebank/ResumeBankHome" or you can use the <reference target=<componentName/serviceName> instead of the URI.

If the EJB reference binding is accessing a stateless session bean that is deployed in the same cell, the URI can be based on cluster/node/server setup, for example:

```
uri="cell/clusters/cluster2/ejb/com/app/resumebank/ResumeBankHome"
uri="cell/nodes/<node_name>/servers/<server_name>/ejb/com/app/resumebank/ResumeBankHome"
```

If the EJB reference binding is accessing a stateless session bean on a different cell (cross cell) or a mixed cell, you need to create a namespace binding, either an enterprise bean or Corba type, in the WebSphere Application Server administrative console and use the name in namespace binding in EJB reference binding URI such as, uri="cell/persistent/<*name_in_namespace_binding*>". For example, uri="cell/persistent/*neworder*" where *neworder* is name in the namespace binding.

SCA EJB Session Bean Binding

Table 26. Unsupported sections of the SCA EJB Session Bean Binding specification

Section	Not supported in Feature Pack for SCA v1.0

2.1 Session Bean Binding Schema	`/binding.ejb/@session-type`
	• Since Feature Pack for SCA does not support conversations, although session-type is set to "stateful", the service still behaves as stateless.
	`/binding.ejb/@uri`
	• Line 91: Feature Pack for SCA only supports the following formats:
	– For EJB2
	`corbaname:iiop:<hostName>:<port>/ NameServiceServerRoot#ejb/ sca/ejbbinding/<componentName>/<serviceName>`
	– For EJB3
	`corbaname:iiop:<hostName>: <port>/NameServiceServerRoot#ejb/sca/ejbbinding/ <componentName>/<serviceName># <serviceInterfaceName>Remote or corbaname:iiop: <hostName>: <port>/NameServiceServerRoot# <serviceInterfaceName>Remote`
	• Line 97: corbaname:rir:#ejb/MyHome

2.3.1 Conversational Nature of Stateful Session Beans	Lines 197-229

Using EJB bindings in SCA applications in a cluster environment

Use this task to learn how to use Enterprise JavaBeans (EJB) bindings that are deployed in Service Component Architecture (SCA) applications in a cluster environment.

Service side

When an SCA service is exposed with a binding.ejb element, the service is exposed through an enterprise bean. During deployment, the SCA runtime generates a session bean for the service that is exposed through EJB binding. The caller of the composite service can invoke this service by accessing the generated EJB.

If the service is exposed through an EJB 2 bean, the EJB is bound at:

`ejb/sca/ejbbinding/`*`component_name/service_name`*

For example:

`ejb/sca/ejbbinding/CompanyComponent/Company`

If the service is exposed through an EJB 3 bean, the EJB is bound at:

`ejb/sca/ejbbinding/`*`component_name/service_name#fullyQualifiedServiceInterfaceName`*`Remote`

For example:

`ejb/sca/ejbbinding/CompanyComponent/Company#com.app.jobbank.CompanyRemote`

The generated EJB for the composite service will be under *profile_root*/installedApps/*cell_name*/*composite_name*.ear/.

Callers need to include client required classes (such as remote or home) of this generated bean in their classpath (or bundle the classes in their JAR file).

Lookup and invoke of this generated service EJB from a clustered environment is the same as lookup and invoke of any EJB in a product clustered setup. Refer to ″Naming considerations in clustered and cross-server environments″ in the *EJB 3.0 application bindings overview* topic.

Reference side

When used on the reference side, the binding.ejb element should specify a URI attribute with values that match the value that is typically used when an EJB client calls the initialContext.lookup() method. The general convention is:

```
"corbaname:iiop:host:port/NameServiceServerRoot#JNDI_name"
```

where *JNDI_name* is the JNDI name of the target EJB.

For example:

```
uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#ejb/session/PriceQuoteSessionFacadeBean"
```

JNDI name syntax differs if the target EJB is an EJB 2 or EJB 3 bean.

When the referred EJB service is in a different cell, the URI might resemble one of the following:

```
uri="corbaname:iiop:localhost:2809/NameServiceServerRoot#cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"
```

or

```
uri="corbaname::/NameServiceServerRoot#cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"
```

or

```
uri="cell/clusters/cluster1/ejb/session/PriceQuoteSessionFacadeBean"
```

if the target EJB is on the same machine but on different cluster.

In advanced scenarios on multiple-server environments, a simpler and more portable way to access the target EJB application from an SCA composite is to set up a namespace binding and use the namespace binding name in the URI attribute of the binding.ejb along with `cell/persistent/`. For example:

```
uri="cell/persistent/PriceQuote"
```

where PriceQuote is the `name` field in the namespace.

The namespace binding can be of type EJB or CORBA based on the advanced scenario.

If the target EJB application which the composite is trying to access is on same cell, but on a different server, node or cluster, configure an EJB namespace binding. You can do this from the administrative console:

1. Click **Environment** → **Naming** → **Name space bindings**.
2. Select the cell scope.
3. Click **New**.
4. On the Specify binding type page, select the **EJB** binding type.

5. On the Specify basic properties page, specify the binding identifier, name in namespace, enterprise bean location such as server cluster or single server (with node), and JNDI name as needed. Use the **Name in name space** field to construct the URI as `cell/persistent/name_in_namespace`.

If the composite is running on a Version 7.0 cell and the target EJB application is running on a Version 6.1 product, configure a CORBA namespace binding with the correct Corbaname URL of the target EJB. Example Corbaname URL syntax is:

`"corbaname:iiop:host:port/NameServiceServerRoot#jndi_name"`

After you configure the namespace binding, use the **Name in name space** field to construct the URI; for example, `uri="cell/persistent/PriceQuote"` where PriceQuote is the value in the **Name in name space** field of the binding.

An advantage of using a namespace binding is, even when the target EJB changes, the composite definition does not need to change. Only the namespace binding needs to change accordingly.

Resolving SCA references

During application assembly or deployment, a reference (a service dependency) is typically resolved to an actual deployed SCA service.

About this task

There are two ways to resolve an Service Component Architecture (SCA) reference:

- Using the @target attribute in order to resolve a reference to a component service within the domain
- Using a binding-specific endpoint using the @uri attribute of the binding element

Resolving an SCA reference using the @target attribute

Use this option when the target service is another SCA service that is in the same domain as the client component, or rather, the component with the reference.

When you have configured the SCA default binding for your SCA service, the @target attribute is typically used to resolve an SCA client reference to the SCA component service. You can also use the @target attribute to resolve an SCA reference when using the SCA non-default bindings. Using this mechanism, the client does not need to know the endpoint of a service that is calculated during run time in order to resolve to it. Whereas a binding-specific endpoint can contain server-specific information such as an Hypertext Transfer Protocol (HTTP) port for a Web service binding, the @target attribute does not require an update when the client-service pair is deployed to a new server with different ports. You can also redeploy a target service within the domain from a single server to a cluster without requiring a change to the reference-side composite definition. If you use this approach, remember that you must use bindings of the same type, meaning that the reference must share a common binding with the service it is targeting.

Resolving an SCA reference using a binding-specific endpoint

You must resolve an SCA reference using a binding-specific endpoint if you invoke non-SCA services over non-default bindings or if you have compatible SCA services that are hosted in another domain.

In general, obtain the binding-specific endpoint from the service provider.

If your target service is another SCA service, see the documentation for configuring the SCA Web service binding, and the SCA EJB binding to learn more about which binding-specific endpoint is used for a given service deployment over a particular binding.

1. Determine from the service provider whether the service that you are consuming is an SCA service within the same domain as your client.
2. Determine the binding that your client uses to consume this service. If the target service is an SCA service, the binding that you use is based on the bindings over which the service is exposed. If the service is not an SCA service, the binding depends on the technology over which the service is provided. For example, services offered over SOAP/HTTP use the SCA Web service binding.
3. If the SCA service is hosted in the same domain as your client, use the @target attribute to resolve a reference to a component service within the domain.

   The following examples demonstrate using the @target attribute. The syntax for the `<reference>` element is the same for the different SCA binding types.

   - SCA default binding

     Target component

     ```
     <component name="TargetComponent">
         <service name="BankService"/>
     </component>
     ```

     Client component

     ```
     <component name="ClientComponent">
         <reference name="myReference" target="TargetComponent"/>
     </component>
     ```

   - SCA Web service binding

     Target component

     ```
     <component name="TargetComponent">
        <service name="BankService">
           <interface.wsdl ....>
           <binding.ws/>
        </service>
     </component>
     ```

     Client component

     ```
     <component name="ClientComponent">
        <reference name="myReference" target="TargetComponent">
            <interface.wsdl ....>
           <binding.ws/>  <!-- The client does not have to worry about endpoint details. -->
        </reference>
     </component>
     ```

4. Resolve the SCA reference by using a binding-specific endpoint if you are invoking non-SCA services over non-default binding or if you have compatible SCA services that are hosted in another domain.

   The following example demonstrates using the SCA Web service binding endpoint for the client component:

   Client component

   ```
   <component name="ClientComponent">
       <reference name="myReference">
                <!-- The exact URL is obtained from a service provider. -->
                <binding.ws uri="http://www.mybank.com:9080/MyBank/AccountService/services>
       </reference>
   </component>
   ```

See the documentation for configuring the SCA Web service binding, and the SCA EJB binding to learn more about binding-specific endpoint resolution for these SCA binding types.

Results

You have identified your SCA client's reference to a target service that it will consume.

Chapter 7. Using JAXB for XML data binding

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified Web services development. JAXB provides the xjc schema compiler, the schemagen schema generator and a runtime framework to support marshalling and unmarshalling of XML documents to and from Java objects.

About this task

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides tooling to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) tooling and implementation within this product. You can develop JAXB objects to use within your JAX-WS applications. You can also use JAXB independently of the JAX-WS programming model as a convenient way to leverage the XML data binding technology to manipulate XML within your Java applications.

JAXB is also the default data binding technology used by Service Component Architecture (SCA) applications. JAXB enables the SCA service implementation side and the SCA client reference side to interact with Java objects without worrying about how the data is transformed into and from XML. JAXB is supported for the binding.sca and binding.ws binding types.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

JAXB provides the xjc schema compiler tool, the schemagen schema generator tool, and a runtime framework. The xjc schema compiler tool enables you to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the schemagen schema generator tool to create the XML schema. After using either the schema compiler or the schema generator command-line tools, you can convert your XML documents both to and from Java objects and use the resulting Java classes to assemble a Web services application.

In addition to using the tools from the command-line, you can invoke these JAXB tools from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask`

Ant task from within the Ant build environment to invoke the xjc schema compiler tool. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

JAXB annotated classes and artifacts contain all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML files and unmarshaling the XML document back to JAXB class instances. The JAXB binding package, `javax.xml.bind`, defines the abstract classes and interfaces that are used directly with content classes. In addition the package defines the marshal and unmarshal APIs.

JAXB 2.1 provides enhancements such as improved compilation support and support for the @XMLSeeAlso annotation. With JAXB 2.1, you can configure the xjc schema compiler so that it does not automatically generate new classes for a particular schema. Similarly, you can configure the schemagen schema generator to not automatically generate a new schema. This enhancement is useful when you are using a common schema and you do not want a new schema generated. JAXB 2.1 also introduces the @XMLSeeAlso annotation that enables JAXB to bind additional Java classes that it might not otherwise know about when binding a Java class with this annotation. This annotation enables JAXB to know about all classes that are potentially involved in marshalling or unmarshalling as it is not always possible or practical to list all of the subclasses of a given Java class. JAX-WS 2.1 also supports the use of the @XMLSeeAlso annotation on a service endpoint interface (SEI) or on a service implementation bean to ensure all of the classes referenced by the annotation are passed to JAXB for processing.

You can optionally use JAXB binding customizations to customize generated JAXB classes by overriding or extending the default JAXB bindings when the default bindings do not meet your business application needs. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes. JAXB supports binding customizations and overrides to the default binding rules that you can make through various ways. For example, you can the overrides inline as annotations in a source schema, as declarations in an external bindings customization file that is used by the JAXB binding compiler, or as Java annotations within Java class files used by the JAXB schema generator. See the JAXB specification for information regarding binding customization options.

Using JAXB, you can manipulate data objects in the following ways:

- Generate an XML schema from a Java class. Use the schema generator schemagen command to generate an XML schema from Java classes.
- Generate Java classes from an XML schema. Use the schema compiler xjc command to create a set of JAXB-annotated Java classes from an XML schema.
- Marshal and unmarshal XML documents. After the mapping between XML schema and Java classes exists, use the JAXB binding runtime to convert XML instance documents to and from Java objects.

Results

You now have JAXB objects that your Java application can use to manipulate XML data.

Using JAXB schemagen tooling to generate an XML schema file from a Java class

Use Java Architecture for XML Binding (JAXB) schemagen tooling to generate an XML schema file from Java classes.

Before you begin

Identify the Java classes or a set of Java objects to map to an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between Java classes and XML schema. XML schema documents describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a bottom-up development approach starting from existing JavaBeans or enterprise beans, use the wsgen tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications or the Service Component Architecture (SCA) representations of your business service interfaces. After the Java artifacts for your application are generated, you can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

You can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime

environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

JAXB 2.1 provides improvements in compilation support to enable you to configure the schemagen schema generator so that it does not automatically generate a new schema. This is helpful if you are using a common schema such as the World Wide Web Consortium (W3C), XML Schema, Web Services Description Language (WSDL), or WS-Addressing and you do not want a new schema generated for a particular package that is referenced. The location attribute on the @XmlSchema annotation causes the schemagen generator to refer to the URI of the existing schema instead of generating a new one.

In addition to using the schemagen tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

Note: When running the schemagen tool, the schema generator does not correctly read the @XmlSchema annotations from the package-info class file to derive targetNamespaces. Instead of using the @XMLSchema annotation, use one of the following methods:

- Provide a package-info.java file with the @XmlSchema; for example:
  ```
  schemagen sample.Address sample\package-info.java
  ```
- Use the @XmlType annotation namespace attribute to specify a namespace; for example:
  ```
  @XmlType(namespace="http://myNameSpace")
  ```

1. Locate the Java source files or Java class files to use in generating an XML schema file. Ensure that all classes referenced by your Java class files are contained in the classpath or are provided to the tool using the`-classpath/-cp` options.
2. Use the JAXB schema generator, schemagen command to generate an XML schema. The schema generator is located in the *app_server_root*\bin\ directory.

   .
   ```
   app_server_root\bin\schemagen.bat myObj1.java myObj2.java
   ```

   ....
   ```
   app_server_root/bin/schemagen.sh myObj1.java myObj2.java
   ```

   .
   ```
   app_server_root/bin/schemagen myObj1.java myObj2.java
   ```

   The parameters, `myObj1.java` and `myObj2.java`, are the names of the Java files containing the data objects. If `myObj1.java` or `myObj2.java` refer to Java classes that are not passed into the schemagen command, you must use the `-cp` option to provide the classpath location for these Java classes.Read about the schemagen command to learn more about this command and additional options that you can specify.
3. (Optional) Use JAXB program annotations defined in the `javax.xml.bind.annotations` package to customize the JAXB XML schema mappings.

4. (Optional) Configure the `location` property on the @XmlSchema annotation to indicate to the schema compiler to use an existing schema rather than generating a new one. For example,

```
@XmlSchema(namespace="foo")
package foo;
@XmlType
class Foo {
@XmlElement Bar zot;
}
@XmlSchema(namespace="bar",
location="http://example.org/test.xsd")
package bar;
@XmlType
class Bar {
...
}
<xs:schema targetNamespace="foo">
<xs:import namespace="bar"
schemaLocation="http://example.org/test.xsd"/>
<xs:complexType name="foo">
<xs:sequence>
<xs:element name="zot" type="bar:Bar" xmlns:bar="bar"/>
</xs:sequence>
</xs:complex
```

   the `location="http://example.org/test.xsd"` indicates the location on the existing schema to the schemagen tool and a new schema is not generated.

Results

Now that you have generated an XML schema file from Java classes, you are ready to marshal and unmarshal the Java objects as XML instance documents.

Note: The **schemagen** command does not differentiate the XML namespace between multiple @XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Example

The following example illustrates how JAXB tooling can generate an XML schema file from an existing Java class, Bookdata.java.

1. Copy the following `Bookdata.java` file to a temporary directory.

```
package generated;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.datatype.XMLGregorianCalendar;
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "bookdata", propOrder = {
    "author",
    "title",
    "genre",
    "price",
    "publishDate",
    "description"
})
public class Bookdata {

    @XmlElement(required = true)
    protected String author;
    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String genre;
    protected float price;
    @XmlElement(name = "publish_date", required = true)
    protected XMLGregorianCalendar publishDate;
    @XmlElement(required = true)
    protected String description;
    @XmlAttribute
    protected String id;

     public String getAuthor() {
        return author;
    }
    public void setAuthor(String value) {
        this.author = value;
    }
    public String getTitle() {
        return title;
    }

     public void setTitle(String value) {
        this.title = value;
    }


    public String getGenre() {
        return genre;
    }

    public void setGenre(String value) {
        this.genre = value;
    }


    public float getPrice() {
        return price;
    }


    public void setPrice(float value) {
        this.price = value;
    }


    public XMLGregorianCalendar getPublishDate() {
        return publishDate;
    }


    public void setPublishDate(XMLGregorianCalendar value) {
        this.publishDate = value;
```

```
    }


    public String getDescription() {
        return description;
    }


    public void setDescription(String value) {
        this.description = value;
    }


    public String getId() {
        return id;
    }


    public void setId(String value) {
        this.id = value;
    }

}
```

2. Open a command prompt.
3. Run the **schemagen** schema generator tool from the directory where you copied the `Bookdata.java` file.

```
app_server_root\bin\schemagen.bat Bookdata.java
```

```
app_server_root/bin/schemagen.sh Bookdata.java
```

```
app_server_root/bin/schemagen Bookdata.java
```

4. The XML schema file, `schema1.xsd` is generated:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="bookdata">
    <xs:sequence>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="genre" type="xs:string"/>
      <xs:element name="price" type="xs:float"/>
      <xs:element name="publish_date" type="xs:anySimpleType"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Refer to the JAXB Reference implementation documentation for additional information about the **schemagen** command.

Using JAXB xjc tooling to generate JAXB classes from an XML schema file

Use Java Architecture for XML Binding (JAXB) xjc tooling to compile an XML schema file into fully annotated Java classes.

Before you begin

Develop or obtain an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a top-down development approach starting with an existing Web Services Description Language (WSDL) file, use the wsimport tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications or the Service Component Architecture (SCA) Java representations of your business service interfaces when starting with a WSDL file. After the Java artifacts for your application are generated, you can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, xjc command-line tool. The resulting annotated Java classes contain all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or other Java applications such as SCA applications for processing XML data.

In addition to using the xjc tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the xjc schema compiler tool.

1. Use the JAXB schema compiler, xjc command to generate JAXB-annotated Java classes. The schema compiler is located in the *app_server_root*\bin\ directory. The schema compiler produces a set of packages containing Java source files and JAXB property files depending on the binding options used for compilation.
2. (Optional) Use custom binding declarations to change the default JAXB mappings. Define binding declarations either in the XML schema file or in a separate bindings file. You can pass custom binding files by using the **-b** option with the xjc command.
3. Compile the generated JAXB objects. To compile generated artifacts, add the Thin Client for JAX-WS with WebSphere Application Server to the classpath.

Results

Now that you have generated JAXB objects, you can write Java applications using the generated JAXB objects and manipulate the XML content through the generated JAXB classes.

Example

The following example illustrates how JAXB tooling can generate Java classes when starting with an existing XML schema file.

1. Copy the following `bookSchema.xsd` schema file to a temporary directory.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="CatalogData">
     <xsd:complexType >
```

```
      <xsd:sequence>
        <xsd:element name="books" type="bookdata" minOccurs="0"
 maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="bookdata">
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="genre" type="xsd:string"/>
      <xsd:element name="price" type="xsd:float"/>
      <xsd:element name="publish_date" type="xsd:dateTime"/>
      <xsd:element name="description" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

2. Open a command prompt.
3. Run the JAXB schema compiler, xjc command from the directory where the schema file is located. The xjc schema compiler tool is located in the *app_server_root*\bin\ directory.

   ```
   app_server_root\bin\xjc.bat bookSchema.xsd
   ```

   ```
   app_server_root/bin/xjc.sh bookSchema.xsd
   ```

   Running the xjc command generates the following JAXB Java files:

   ```
   generated\Bookdata.java
   generated\CatalogdData.java
   generated\ObjectFactory.java
   ```

4. Use the generated JAXB objects within a Java application to manipulate XML content through the generated JAXB classes.

Refer to the JAXB 2.0 Reference implementation documentation for additional information about the xjc command.

Chapter 8. SCA application package deployment

The IBM Feature Pack for Service Component Architecture (SCA) supports deployment of many types of SCA artifacts as composition units of business-level applications. Typical artifacts include Java archive (JAR) files, compressed (ZIP) files, and Web application archive (WAR) files.

Details about deployment of SCA artifacts in the Feature Pack for SCA follow.
- Deployment of JAR or ZIP files
- Deployment of WAR files
- Notes and limitations

Deployment of JAR or ZIP files

- The product supports one composite file for each package. The product determines which composite file to support using the following process:
  1. The SCA deployment extension looks for the `META-INF/sca-contribution.xml` file, gets the name of each deployable composite defined in the file, and uses QName values to find the actual composite file names under any directory for that composite. If more than one composite is found in the `sca-contribution.xml` file, you can select the composite to deploy.
  2. If there is no `META-INF/sca-contribution.xml` file defined, the SCA deployment extension looks for a composite file in the `META-INF/sca-deployables` directory.
- The product validates SCA composites for inconsistencies with SCA specifications.

  One specification requirement is that the names of top-level components must be unique. Thus, the product validates top-level component name uniqueness.

  Tip: Top-level components are also called domain components, with the top-level or domain typically the cell on multiple-server installations and the server scope on single-server installations.

  The product validates all composite files in a JAR or ZIP file, regardless of the file location in the archive or whether the `sca-contribution.xml` file references the composite file. The product does not validate all services and references.

  The product writes warning and error messages resulting from the validation tests to the `SystemOut.log` file. Read the log file to learn about inconsistencies with specifications in your SCA composites.
- The product uses the following process for QName resolution:
  – The product uses the QName to resolve composite files included in the top-level composite that use the element. For example, the `<include name="mynamespace:MyService"/>` statement looks for a composite file whose composite name is `MyService` and whose targetNameSpace is `mynamespace`. The following rules apply:
    - **name**: Use the outer composite.
    - **namespace declarations**: Merged into the outer composite.
    - **targetNamespace**: Use the outer composite (must be the same).
    - **local**: Use the composite (preferably the same but not required).
    - **requires(intents) and policySets**: Must be compatible, and aggregated into the outer composite.

Deployable composite files must have name and targetNamespace values. The name and targetNamespace values are combined to form the QName of a composite file.

  - When a composite is used as a component implementation in the top-level composite, the composite is also resolved using the QName. For example, the `<implementation.composite name="mynamespace:MyComposite"/>` statement causes the product administration to look for a composite file whose composite name is `MyComposite` and whose targetNamespace is `mynamespace`.

- A JAR file can contain other JAR files at the top level. The contained JAR files are available on the classpath. However, any archives inside those JAR files are not available. The product supports one level of embedded JAR files.

Deployment of WAR files

- A composite file in a WAR file must be named `default.composite`. A composite file that is not in a WAR file can have any name.
- The `default.composite` composite file must be inside a WAR file in the `META-INF/sca-deployables` directory.
- The `META-INF/sca-deployables` directory must contain no more than one composite file. If there is more than one composite file in the `META-INF/sca-deployables` directory, then the product returns a validation error and stops the WAR file deployment.

  However, you can place other composite files in directories other than `META-INF/sca-deployables`, and reference those composite files in the top-level composite under the `META-INF/sca-deployables` directory.
- The product does not support having a `sca-contribution.xml` file inside the WAR file under the `META-INF` directory. If the product finds a `sca-contribution.xml` file, then the product returns a validation error and stops the WAR file deployment.

Notes and limitations

- The product does not provide administration console pages for configuring contributions.
- If you import a package or namepace from a different contribution, or JAR (`contribution.xml`), you might need to import that contribution as an asset before importing your own asset.

  For example, suppose your Contribution A imports a JAR file from Contribution B. You might need to import Contribution B and then Contribution A as assets. Contribution A depends on Contribution B so you must import Contribution B before importing Contribution A.
- You cannot use a local interface across a class loader boundary. Use a remotable interface to cross a class loader boundary.

Chapter 9. Creating SCA business-level applications

You can create an empty business-level application and then add Service Component Architecture (SCA) assets, shared libraries, business-level applications, and other artifacts as composition units to the empty business-level application.

Before you begin

Configure the target application server. You must deploy SCA composite assets of a business-level application to a Version 7.0 server or cluster that is enabled for the Feature Pack for SCA.

Optionally, determine what assets or other files that you want to add to your business-level application and whether your application files can run on your deployment targets.

About this task

You can create business-level applications using the administrative console, the wsadmin tool, or programming.

Note: You create SCA business-level applications the same way as for non-SCA business-level applications. However, when you use an SCA asset in a business-level application, function that applies only to applications that use SCA composites becomes available. For example, you can access administrative console pages that apply only to applications that use SCA composites. The Feature Pack for SCA extends the business-level application functionality.

1. Select a way to create your business-level application.

Table 27. Ways to create SCA business-level applications

Option	Method
Administrative console business-level application creation wizard See "Creating SCA business-level applications with the console" on page 92.	Click **Applications** → **New Application** → **New Business Level Application** and follow instructions in the wizard. For example use of the console to create a business-level application that has an SCA asset, see "Example: Creating an SCA business-level application with the console" on page 116.

2. Create your business-level application using the administrative console, wsadmin, or programming.
3. Save the changes to your administrative configuration.

   When saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Results

The name of the application is shown in the list on the Business-level applications page.

What to do next

After you create a business-level application, you can do the following to add composition units to it:

1. Import any SCA or other assets needed by your business-level application.
2. Add assets, shared libraries, or other business-level applications as composition units.

   When you add an asset, you must specify a target server or cluster that is enabled with the Feature Pack for SCA.
3. Save the changes to your administrative configuration.
4. Start the business-level application.

If the application does not run as desired, edit the application configuration, then save and run it again.

If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again. If SCA composite assets do not start, ensure that each asset is mapped to a deployment target that is enabled for the Feature Pack for SCA.

If an asset composition unit uses an Enterprise JavaBean (EJB) binding and does not start because it has a `non-WebSphere target of "null"`, delete the asset composition unit and add it again to the business-level application. Specify a target enabled for the Feature Pack for SCA when you add the asset to the business-level application. You cannot change the target after deployment.

If the `META-INF/sca-deployables` directory has multiple SCA composite files and the application does not start because the product `cannot obtain the CompUnitInfoLoader value`, place only the file that contains the composite in the `META-INF/sca-deployables` directory. You can place the other composite files anywhere else within the archive.

If the SCA application uses security, the target server or cluster must be in the global security domain.

In multiple-node environments, synchronize the nodes after you save changes to the target before starting the business-level application.

Creating SCA business-level applications with the console

You can create an empty business-level application and then add Service Component Architecture (SCA) assets, shared libraries, or business-level applications as composition units to the empty business-level application.

Before you begin

Configure the target application server. You must deploy SCA composite assets of a business-level application to a Version 7.0 server or cluster that is enabled for the Feature Pack for SCA.

Also, determine an application name. Optionally, determine which assets, shared libraries, or business-level applications that the new business-level application needs.

About this task

You can create a business-level application that has SCA assets using the administrative console. Alternatively, you can use the wsadmin scripting tool or programming.

You can add an asset or shared library composition unit to multiple business-level applications. However, each composition unit for the same asset must have a unique composition unit name. You can add a business-level application composition unit to more than one business-level application.

1. Create an empty business-level application.
   a. Click **Applications** → **New Application** → **New Business-level Application**.
   b. On the New business-level application page, specify a unique name for the application and a description, and then click **Apply**.
   c. On the business-level application settings page, click **Save**.

   The name and description are shown in the list of applications on the Business-level applications page. Because the application is empty, its status is `Unknown`.
2. Add one SCA asset to your business-level application. The product adds the asset as a composition unit of your business-level application.
   a. Import the SCA asset.
   b. Go to the business-level application settings page.

      Click **Applications** → **Application Types** → **Business-level applications** → ***application_name***.
   c. On the business-level application settings page, specify the type of composition unit to add.

      Although you can add an asset, shared library, or business-level application to your business-level application, the logic is in your SCA asset. Add the SCA asset as a composition unit.

      Under **Deployed assets**, click **Add** → **Add Asset**.
   d. On the Add page, select one unit from the list of available units, and then click **Continue**.

      On the Add page, you might be able to select multiple deployable SCA composites. However, the Feature Pack for SCA supports deploying only one deployable composite at a time. Select only one unit and click **Continue**. If you select multiple units, the product deploys only one of those units.
   e. On the Set options page, change the composition unit settings as needed, and then click **Next**.

      This page is not shown if you have multiple deployable unit assets.
   f. On the Map composition unit to a target page, specify a target server that is enabled for the Feature Pack for SCA, and then click **Next**.

      The target server can be an existing cluster. To map the composition unit to a cluster, select the existing cluster from the **Available** list, click **Add**, and then click **Next**. The cluster name is shown in the **Current targets** list as `WebSphere:cluster=`*`cluster_name`*.

      If you are adding an SCA asset that uses security, specify a target server that is in the global security domain.

      This page is not shown when you add a business-level application.

g. On the Relationship options page, specify composition unit relationship options.

   Specify a relationship if a deployable unit depends on another asset deployed as a shared library in order to run. This page is shown only for SCA assets that have multiple deployable or composition units.

h. On the Map security roles to users or groups page, specify security roles for users or groups as needed, and then click **Next**.

   This page is only shown for SCA assets that use security.

i. On the Map RunAs roles to users page, map a user identity and password to RunAs roles as needed, and then click **Next**.

   This page is only shown for SCA assets that use security.

j. On the Map virtual host page, specify a virtual host that hosts Web services for each SCA composite, and then click **Next**. By default, composites map to `default_host`.

   This page is only shown for SCA assets that contain a Web service binding.

k. On the Attach policy set page, attach a policy set and assign policy set bindings as needed, and then click **Next**.

   This page is only shown for SCA assets that use Web services.

l. On the Summary page, click **Finish**. Several messages are displayed, indicating whether the product adds the unit to the business-level application successfully. A message having the format `Completed res=[WebSphere:cuname=`*`unit_name`*`]` indicates that the addition is successful. Click **Manage application**.

   If the product adds the unit successfully, the name of the unit is shown in a list of deployed assets on the business-level application settings page.

   If the unit addition is not successful, read the messages and add the unit again. Correct the problems noted in the messages.

m. On the Adding composition unit to the business-level application page, click **Save**.

3. Optional: Add one or more assets, shared libraries, or business-level applications to your business-level application.

   Repeat Step 2 to add another asset or to add a shared library or business-level application.

Results

A business-level application that contains the specified composition units.

What to do next

After you create the application, save the changes to your configuration and start the application as needed.

If a composite asset is deployed to a non-SCA server or cluster, the SCA composite does not start. You must deploy SCA assets to servers or clusters that are enabled for the feature pack.

Map virtual host settings for SCA composites

Use this page to map Service Component Architecture (SCA) composites that use a Web service binding to a virtual host. You must map the composites to the virtual host that hosts the Web services.

This administrative console page displays in the business-level application creation and update wizards. To view the Map virtual host page, the asset that you add to a business-level application must contain a Web service binding. To view this page, do the following:

1. Import an asset that contains a Web service binding.
2. Create a business-level application to which to add the asset.
3. Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → **Add** → **Add Asset**.
4. On the Add composition unit page, select the asset that contains a Web service binding, and click **Continue**.
5. On the Set options page, change the settings as needed and click **Next**.
6. On the Map composition unit to a target page, specify target servers as needed and click **Next**.
7. On the Define relationship with existing composition units page, change the settings as needed and click **Next**.
8. Continue changing settings as needed and click **Next** on any other pages until the Map virtual host page is displayed in the wizard.

Composite Name

Specifies the name of the composite that uses a Web service binding in the SCA artifact.

Virtual Host

Specifies a virtual host to associate with the composite.

Select the virtual host that hosts the Web services for the composite. By default, the product associates a component with the `default_host` virtual host.

Attach policy set settings

Use this page to attach a policy set and assign policy set bindings for the composite defined in a Service Component Architecture (SCA) application.

This administrative console page displays in the Create new business-level application wizard. To have the Attach policy set page in the wizard, the SCA component in the asset that you add to a business-level application must use a Web service binding, `binding.ws`, and the composite file or annotation must specify the intents or policy sets. To view this page, do the following:

1. Import an asset that uses a Web service binding and a composite file or annotation that specifies the intents or policy sets.
2. Create a business-level application to which to add the asset.
3. Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → **Add** → **Add Asset**.
4. On the Add composition unit page, select the asset that uses a Web service binding, and click **Continue**.
5. On the Set options page, change the settings as needed and click **Next**.
6. On the Map composition unit to a target page, specify target servers as needed and click **Next**.
7. On the Define relationship with existing composition units page, change the settings as needed and click **Next**.
8. Continue changing settings as needed and click **Next** on any other pages until the Attach policy set page is displayed in the wizard.

To attach or detach a policy set or binding, do the following:

1. Select a composite, component, service, reference, or binding from **Name**. The **Name** list is nested, indicating parent-child relationships. When you select a parent, the children are automatically selected.
2. Click the desired button.

Table 28. Button descriptions

Button	Resulting action
Attach	Attaches a policy set to the selected composite, component, service, reference, or binding. When the **Include default policy sets** option is not enabled, the options for this button contain user-created policy sets only. When the **Include default policy sets** option is not enabled and no user-created policy sets exist, then there are no button options. You can select **Include default policy sets** to display the default policy set options. When the **Include default policy sets** option is enabled, the options for this button include both default policy sets and any user-created policy sets. To attach a policy set, select a composite, component, service, reference, or binding from **Name** and click **Attach** → ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected composite, component, service, reference, or binding.
Assign Service Policy Set Binding	Assigns a service policy set binding to the selected composite, component, service, reference, or binding. There are two default options: **Default** specifies to assign the default service policy set binding. **Provider Sample** specifies to assign a policy set binding that is provided with the product to the service. If you are deploying the composition unit to a server or cluster that belongs to a security domain, the list of policy set bindings consists of bindings that have been defined in the security domain to which the composition unit is being deployed.
Assign Reference Policy Set Binding	Assigns a reference policy set binding to the selected composite, component, service, reference, or binding. There are two default options: **Default** specifies to assign the default reference policy set binding. **Client Sample** specifies to assign a policy set binding that is provided with the product to the reference. If you are deploying the composition unit to a server or cluster that belongs to a security domain, the list of policy set bindings consists of bindings that have been defined in the security domain to which the composition unit is being deployed.

Include default policy sets

Specifies whether to include default policy sets. Default policy sets specify common quality of service (QoS) behavior for generic message format.

Before selecting this option, determine whether the default policy sets provide adequate QoS characteristics for your services.

By default, this option is not enabled.

Name
Specifies a composite, component, service, reference, or binding in the artifact.

The **Name** list is nested, indicating parent-child relationships. When you select a parent, the children are automatically selected.

Intents
Specifies the aggregate of the intents from the composite file and the annotations. SCA intents are used to describe the abstract policy requirements of a component.

The intents shown include any intents inherited from a parent.

Matched Policy Sets
Specifies policy sets that potentially satisfy the intents.

You can include default policy sets by enabling the **Include default policy sets** check box. To exclude default policy sets, deselect the check box.

Attached Policy Set
Specifies attached policy sets. If no value is shown, then the composite, component, service, reference, or binding is not attached to a policy set.

To attach a policy set, select a composite, component, service, reference, or binding and click an **Attach** option.

To detach a policy set, use **Detach Policy Set**. You can detach any policy set, including pre-attached policy sets.

Policy Set Binding
Specifies service and reference policy set bindings. If no value is shown, then the composite, component, service, reference, or binding is not assigned to a policy set binding.

To assign a policy set binding, select a composite, component, service, reference, or binding and click an **Assign Service Policy Set Binding** or **Assign Reference Policy Set Binding** option.

To reset the bindings, select the **Default** option. For example, select **Assign Service Policy Set Binding** → **Default** or **Assign Reference Policy Set Binding** → **Default**.

Map security roles to users or groups collection for SCA composites
Use this page to view and manage mappings of security roles to users and groups that are used with the Service Component Architecture (SCA) composites.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → **Map security roles to users or groups**. This page is the same as the Map security roles to users or groups page in the Create new business-level application wizard. To view this page, your composition unit must support SCA security.

Different roles can have different security authorizations. Mapping users or groups to a role authorizes those users or groups to access applications defined by the role. Users, groups, and roles are defined when an application is installed or configured.

To map a role to a user or group, enable the Select check box beside the role name in the list and click a button. On the displayed page, specify one or more users or groups to map to the role.

Table 29. Button descriptions

Button	Resulting action
Map Users	Displays the Map users or groups page on which you can specify the users to have the selected security role.
Map Groups	Displays the Map users or groups page on which you can specify the groups to have the selected security role.
Map Special Subjects	Maps special subjects according to the option that you select: **None** specifies to map none of the special subjects to the role. **All Authenticated in Application's Realm** specifies to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role. **All Authenticated in Trusted Realms** specifies to map all of the authenticated users in the trusted realms to a specified role. This option gives all authenticated users who belong to the user registry access to the application's realm and all authenticated users who belong to user registries access to realms which are trusted by the current security domain. **Everyone** specifies to map everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.

Role

Specifies a security role.

Special Subjects

Specifies which special subjects are mapped to the security role. This option applies only when an application uses multiple realms.

None Specifies to map none of the special subjects to the role.

All Authenticated in Application's Realm
Specifies to map all of the authenticated users to a specified role. When you map all authenticated users to a specified role, all of the valid users in the current registry who have been authenticated can access resources that are protected by this role.

All Authenticated in Trusted Realms
Specifies to map all of the authenticated users in the trusted realms to a specified role. All authenticated users who belong to the user registry that is mapped to the application's realm and all authenticated users who belong to user registries that are mapped to realms which are trusted by the current security domain are successfully authorized.

Everyone
Specifies to map everyone to a specified role. When you map everyone to a role, anyone can access the resources that are protected by this role and, essentially, there is no security.

To change the value, select the role, click **Map Special Subjects**, and select an option.

Users
Lists the users that are mapped to the specified role within this application.

Users from the non-default realm are displayed as *user@realm*.

Groups
Lists the groups that are mapped to this specified role within this application.

Map RunAs roles to users collection for SCA composites
Use this page to map a specified user identity and password to a RunAs role for a Service Component Architecture (SCA) composite. This page enables you to specify application-specific privileges for individual users to run specific tasks using another user identity.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → **Map RunAs roles to users**. This page is the same as the Map RunAs roles to users page in the Create new business-level application wizard.

To view this page, the components in your composition unit must contain predefined RunAs roles and support SCA security. RunAs roles are used by components that need to run as a particular role for recognition while interacting with another component.

Username
Specifies a user name for the RunAs role user.

This user already maps to the role specified in the Mapping users and groups to roles page. You can map the user to its appropriate role by either mapping the user to that role directly or mapping a group that contains the user to that role. After you specify the user name and password for the user and select a RunAs role, click **Apply**.

Password
Specifies the password for the RunAs user.

Role
Specifies a security role for a user within this application.

The authorization policy is only enforced when security is enabled.

User
Lists the user that is mapped to the specified role within this application.

Composition unit settings for SCA composites

Use this page to view composition unit settings and to change the properties of a Service Component Architecture (SCA) composite.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_name*. The deployed asset is a composition unit of the business-level application.

To view this page, your composition unit must support SCA.

Name

Specifies a logical name for the composition unit. You cannot change the name on this page.

Description

Specifies a description for the composition unit.

Backing identifier

Specifies a unique identifier for a composition unit that is registered in the application management domain.

The identifier has the format: `WebSphere:`*`unit_type`*`name=`*`unit_name`*. For example, for the `MyApp.jar` asset, the backing identifier might be `WebSphere:assetname=MyApp.jar`.

You cannot change the identifier on this page.

Data type	String
Units	Configuration unit identifier

Starting weight

Specifies the order in which composition units are started when the server starts. The starting weight is like the startup order. The composition unit with the lowest starting weight is started first.

The value that you set for **Starting weight** determines the importance or weight of a composition unit within the business level application. For example, for the most important composition unit within a business-level application, specify `1` for **Starting weight**. For the next most important composition unit within the business-level application, specify `2` for **Starting weight**, and so on.

Data type	Integer
Default	1
Range	0 to 2147483647

Start on distribution

Specifies whether to start the composition unit when the product distributes the composition unit to other locations.

The default is not to start the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

Data type	Boolean

Default false

Recycle behavior on update

Specifies whether the product restarts the composition unit after the composition unit is updated.

The default is to restart the composition unit after partial updating of the composition unit.

This setting applies to asset or shared library composition units. This setting does not apply when the composition unit is a business-level application.

Table 30. Option descriptions

Option	Description
ALL	Restarts the composition unit after the entire composition unit is updated
DEFAULT	Restarts the composition unit after part of the composition unit is updated
NONE	Does not restart the composition unit after the composition unit is updated

Target mapping

Specifies one or more current targets for the composition unit.

To change the existing deployment targets, click **Modify Targets** and select different deployment targets from the list of available clusters and servers.

SCA composite components

Specifies the component names and component implementations of SCA composites in the application.

Table 31. Column descriptions

Column	Description
Component Name	Specifies the name of a component associated with the SCA composite.
Component Implementation	Specifies the name of the class or code implementing the component.

`None` indicates that the SCA composite does not have defined components.

SCA composite properties

Specifies the names and values of SCA composite properties in the application.

Table 32. Column descriptions

Column	Description
Property Name	Specifies the name of an SCA composite property.
Property Value	Specifies the value of the property.

`None` indicates that the SCA composite does not have defined name-value properties.

SCA composite wires

Specifies the sources and targets of wires in the SCA composite.

Table 33. Column descriptions

Column	Description
Wire Source	Specifies the source of a wire in the SCA composite.
Wire Target	Specifies the target of the wire.

`None` indicates that the SCA composite does not have defined wires.

Provide HTTP endpoint URL information settings for SCA composites

Use this page to specify endpoint Universal Resource Locator (URL) prefix information for Service Component Architecture (SCA) composites accessed by Web service bindings. The information is used to form complete endpoint addresses.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → **Provide HTTP endpoint URL information**.

SCA HTTP URL Prefixes

Specifies whether to use an existing default URL prefix or a custom endpoint URL prefix for SCA composites that are accessed by Hypertext Transfer Protocol (HTTP) for service endpoints.

Options to specify a URL prefix include the following:

Select default HTTP URL prefix
To specify an existing default endpoint URL prefix, choose **Select default HTTP URL prefix** and use the menu to select either the secure HTTPS or unsecure HTTP endpoint URL value.

Select custom HTTP URL prefix
You use a custom endpoint URL prefix when the service has a proxied front end. The endpoint URL prefixes are those of the proxy server. You must specify proxied endpoints when deploying services that use the Web service binding in a clustered configuration.

To specify a custom endpoint URL prefix, do the following:

1. Choose **Select custom HTTP URL prefix**
2. In the field, specify both a secure and an unsecure custom endpoint URL value. Separate each URL by a space. For example, specify:

   ```
   http://myHost:9081 https://myHost:9044
   ```

   For each endpoint URL prefix, the format is *protocol*`://`*host_name*`:`*port_number*. Specify the protocol (either `http` or `https`), *host_name*, and *port_number* to use in the endpoint URL.
3. Click **Apply**.

SCA composite component settings

Use this page to view and edit the attributes associated with a Service Component Architecture (SCA) component.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_name* → *SCA_composite_component_name*.

Components are configured instances of implementations. Components provide and consume services. More than one component can use and configure the same implementation, where each component configures the implementation differently. For example each component might configure a reference of the same implementation to consume a different service.

An implementation defines the aspects configurable by a component in the form of a component type. The component type is in effect a description of the contract honored by the implementation.

A reference represents a requirement that the implementation has on a service provided by another component.

Component name
Specifies the component name of the attribute.

Implementation
Specifies the Java class or configuration file that contains the component implementation.

Type
Specifies the type of attribute. In this case, the type is `Component`.

SCA component services
Specifies the names of the services.

SCA component references
Specifies the names and targets of component references. You can edit the reference target for customization.

SCA component properties
Specifies the **Property Input Source** and **Property Value** for each property.

Options for **Property Input Source** include the following:

- **XPath** indicates the source attribute of the property.
- **File** indicates the file attribute of the property.
- **Value** indicates the property element value.

SCA component reference settings
Use this page to view and edit the attributes associated with a Service Component Architecture (SCA) component reference.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → *SCA_composite_component_name* → *reference_name*.

SCA component references within an implementation represent links to services the implementation uses that must be provided by other components in the SCA system. For a composite, you can wire references of components within the

composite (component references) to references of the composite (composite references), indicating that the component references must be resolved by services outside the composite.

References use bindings to describe the access methods used to invoke the services.

Under **Additional Properties**, click **View domain** to view a list of services available in the current cell or domain. This can be helpful when updating the **Target** setting value, for example.

Reference name
Specifies the reference name of the attribute.

Type
Specifies the type of attribute. In this case, it is `Reference`.

Target
Specifies one or more target service uniform resource identifiers (URIs), depending on the multiplicity setting. Each target wires the reference to a component service that resolves the reference. Targets can contain a list of targets separated by a space, in the form **target1 target2**.

Binding
Specifies the URI of the binding.

Supported bindings include the SCA default binding, enterprise bean (EJB) binding, and Web service binding.

SCA component service settings
Use this page to view and edit the attributes associated with a component service.

To view this administrative console page, click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***deployed_asset_composition_unit_name*** → ***SCA_composite_component_name*** → ***service_name***.

Services are used to publish services provided by implementations, so that they are addressable by other components.

A service published by a component can be provided by a service of a component defined within the component, or it can be provided by a component reference. The latter case allows the republication of a service with a new address or new bindings.

Service name
Specifies the service name of the attribute.

Type
Specifies the type of attribute. In this case, `Service`.

Binding
Specifies the uniform resource identifier (URI) of the binding.

Supported bindings include the SCA default binding, enterprise bean (EJB) binding, and Web service binding.

Work manager JNDI name

Specifies the Java Naming and Directory Interface (JNDI) name of the work manager.

Service provider policy sets and bindings collection for SCA composites

Use this page to attach and detach policy sets to a composition unit, a service provider, its endpoints, or operations of a Service Component Architecture (SCA) composite. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service provider can share its current policy configuration.

To view this administrative console page, your composition unit must use Web services and support SCA. Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → **Service provider policy sets and bindings** .

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

To attach or detach a policy set or binding, do the following:

1. Select a composition unit, service, endpoint, or operation. The **Composition unit/Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 34. Button descriptions

Button	Resulting action
Attach	Attaches a policy set to the selected composition unit, service, endpoint, or operation. To attach a policy set, select a unit, service, endpoint, or operation and click **Attach** → ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected composition unit, service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.

Table 34. Button descriptions (continued)

Button	Resulting action
Assign Binding	Assigns a policy set binding to the selected composition unit, service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected composition unit, service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this **Assign Binding** action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Opens a page on which you can create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Provider Sample** Specifies to use a sample binding provided with the artifact.

Composition unit/Service/Endpoint/Operation

Specifies the name of the composition unit and the associated service providers, endpoints or operations.

The Composition unit/Service/Endpoint/Operation column lists the composition unit and the service providers, endpoints, or operations that the composition unit contains.

Attached Policy Set

Specifies the policy set that is attached to a composition unit, service provider, endpoint, or operation.

The Attached Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name*** **(inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding

Specifies the binding configuration that is available for a service provider, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- *Binding_name* or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- *Binding_name* **(inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

References policy sets and bindings collection for SCA composites

Use this page to attach and detach policy sets to a composition unit, a service reference, its endpoints, or operations of a Service Component Architecture (SCA) composite. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service reference can share its current policy configuration.

To view this administrative console page, your composition unit must use Web services and support SCA. Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *deployed_asset_composition_unit_name* → **References policy sets and bindings**.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

To attach or detach a policy set or binding, do the following:

1. Select a composition unit, service, endpoint, or operation. The **Composition unit/Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 35. Button descriptions

Button	Resulting action
Attach Client Policy Set	Attaches a client policy set to the selected composition unit, service, endpoint, or operation. To attach a policy set, select a composition unit, service, endpoint, or operation and click **Attach Client Policy Set** → *policy_set_option*. To close the menu list, click **Attach Client Policy Set**.

Table 35. Button descriptions (continued)

Button	Resulting action
Detach Client Policy Set	Detaches a client policy set from the selected composition unit, service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.
Assign Binding	Assigns a policy set binding to the selected composition unit, service, endpoint, or operation. There are three options: **Default** Specifies the default binding for the selected composition unit, service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this **Assign Binding** action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Opens a page on which you can create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Client Sample** Specifies to use a sample binding provided with the artifact.

Composition unit/Service/Endpoint/Operation

Specifies the name of the composition unit and the associated service references, endpoints or operations.

The Composition unit/Service/Endpoint/Operation column lists the service composition unit and the service references, endpoints, or operations that the composition unit contains.

Attached Client Policy Set

Specifies the policy set that is attached to a composition unit, service reference, endpoint, or operation.

The Attached Client Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- *Policy_set_name*. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- *Policy_set_name* **(inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding

Specifies the binding configuration that is available for a service reference, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- *Binding_name* or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- *Binding_name* **(inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

SCA service provider settings

Use this page to manage policy sets for a Service Component Architecture (SCA) Web service provider. You can attach and detach policy sets to a service provider, its endpoints, or operations. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service provider can share its current policy configuration.

To view this administrative console page, your composition unit must use Web services and support SCA. Click **Services** → **Service providers** → *service_provider_name*.

Service provider

Specifies the full QName of the service provider. The QName must be in a format that supports the Java class `javax.xml.namespace.QName`.

For the SCA sample business-level application HelloWorldAsync, the service provider name resembles the following:

```
{http://websphere.ibm.com/HelloWorldServiceComponent/HelloWorldService}HelloWorldService
```

Policy Set Attachments

Specifies the attached policy sets and assigned bindings for services, endpoints, or operations in the service provider.

To attach or detach a policy set or to assign bindings with system-specific configurations, do the following:

1. Select a service, endpoint, or operation. The **Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 36. Button descriptions

Button	Resulting action
Attach	Attaches a policy set to the selected service, endpoint, or operation. To attach a policy set, select a service, endpoint, or operation and click **Attach** → ***policy_set_option***. To close the menu list, click **Attach**.
Detach Policy Set	Detaches a policy set from the selected service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.
Assign Binding	Assigns a policy set binding to the selected service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this **Assign Binding** action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Opens a page on which you can create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Provider Sample** Specifies to use a sample binding provided with the artifact.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Service/Endpoint/Operation
Specifies the name of the service and the associated service providers, endpoints or operations.

The Service/Endpoint/Operation column lists the service and the service providers, endpoints, or operations that the service contains.

Attached Policy Set
Specifies the policy set that is attached to a service provider, endpoint, or operation.

The Attached Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.
- ***Policy_set_name*** **(inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding
Specifies the binding configuration that is available for a service provider, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name*** **(inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

About policy set bindings

In this release, there are two types of bindings: application-specific bindings and general bindings. Composition units can use both application-specific bindings and general bindings.

Application-specific bindings

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to, and constrained by, the characteristics of the defined policy. Application-specific bindings can provide configuration for advanced policy

requirements such as multiple signatures; however, these bindings are reusable only within an application. Also, application-specific bindings have very limited reuse across policy sets.

When you create an application-specific binding for a policy set attachment, the binding begins in a completely unconfigured state. You must add each policy, such as WS-Security or HTTP transport, that you want to override the default binding, and fully configure the bindings for each policy that you add. For WS-Security policy, some high level configuration attributes such as TokenConsumer, TokenGenerator, SigningInfo, or EncryptionInfo might be obtained from the default bindings if they are not configured in the application-specific bindings.

For service providers, you can create application-specific bindings only by selecting **Assign Binding** → **New Application Specific Binding**, on the Service providers policy sets and bindings collection page, for service provider resources that have an attached policy set. Similarly, for service clients, you can create application-specific bindings only by selecting **Assign Binding** → **New Application Specific Binding**, on the Service clients policy sets and bindings collection page, for service client resources that have an attached policy set.

General bindings

You can configure general bindings to be used across a range of policy sets and they can be reused across applications and for trust service attachments. Although general bindings are highly reusable, they cannot provide configuration for advanced policy requirements such as multiple signatures. There are two types of general bindings: general provider policy set bindings and general client policy set bindings.

You can create general provider policy set bindings by clicking **Services** → **Policy sets** → **General provider policy set bindings** → **New** in the general provider policy sets panel, or by clicking **Services** → **Policy sets > General client policy set bindings** → **New** in the general client policy set and bindings panel. For details about defining and managing service client or provider bindings, see the related links. General provider policy set bindings might also be used for trust service attachments.

SCA service client settings

Use this page to manage policy sets for a Service Component Architecture (SCA) Web service client. You can attach and detach policy sets to a service reference, its endpoints, or operations. You can select the default bindings, create new application-specific bindings, or use bindings that you created for an attached policy set. You can view or change whether the service reference can share its current policy configuration.

To view this administrative console page, your composition unit must use Web services and support SCA. Click **Services** → **Service clients** → *service_client_name*.

Service client

Specifies the full QName of the service client. The QName must be in a format that supports the Java class `javax.xml.namespace.QName`.

For the SCA sample business-level application HelloWorldAsync, the service client name resembles the following:

```
{http://websphere.ibm.com/HelloWorldServiceComponent/HelloWorldService}HelloWorldCallbackService
```

This SCA application has the product Web service namespace, `http://websphere.ibm.com/`, and the service name in its service client name.

Policy Set Attachments

Specifies the attached policy sets and assigned bindings for services, endpoints, or operations in the service client.

To attach or detach a policy set or to assign bindings with system-specific configurations, do the following:

1. Select a service, endpoint, or operation. The **Service/Endpoint/Operation** list is nested, indicating parent-child relationships.
2. Click the desired button.

Table 37. Button descriptions

Button	Resulting action
Attach Client Policy Set	Attaches a client policy set to the selected service, endpoint, or operation. To attach a policy set, select a service, endpoint, or operation and click **Attach Client Policy Set** → ***policy_set_option***. To close the menu list, click **Attach Client Policy Set**.
Detach Client Policy Set	Detaches a client policy set from the selected service, endpoint, or operation. After the policy set is detached, if there is no policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays **None** and the **Binding** column displays **Not applicable**. If there is a policy set attached to an upper-level service resource, the **Attached Client Policy Set** column displays ***policy_set_name*** **(inherited)** and the binding used for the upper-level attachment is applied. The binding name is displayed followed by **(inherited)**.

Table 37. Button descriptions (continued)

Button	Resulting action
Assign Binding	Assigns a policy set binding to the selected service, endpoint, or operation. The options include the following: **Default** Specifies the default binding for the selected service, endpoint, or operation. You can specify client and provider default bindings to be used at the cell level or global security domain level, for a particular server, or for a security domain. The default bindings are used when an application-specific binding has not been assigned to the attachment. When you attach a policy set to a service resource, the binding is initially set to the default. If you do not specifically assign a binding to the attachment point using this **Assign Binding** action, the default specified at the nearest scope is used. For any policy set attachment, the run time checks to see if the attachment includes a binding. If so, it uses that binding. If not, the run time checks in the following order and uses the first available default binding: 1. Default general bindings for the server 2. Default general bindings for the domain in which the server resides 3. Default general bindings for the global security domain **New Application Specific Binding** Opens a page on which you can create a new application-specific binding for the policy set attachments. The new binding you create is used for the selected resources. If you select more than one resource, ensure that all selected resources have the same policy set attached. **Client Sample** Specifies to use a sample binding provided with the artifact.

Depending on your assigned security role when security is enabled, you might not have access to text entry fields or buttons to create or edit configuration data. Review the administrative roles documentation to learn more about the valid roles for the application server.

Service/Endpoint/Operation

Specifies the name of the service and the associated service references, endpoints or operations.

The Service/Endpoint/Operation column lists the service and the service references, endpoints, or operations that the service contains.

Attached Client Policy Set

Specifies the policy set that is attached to a service reference, endpoint, or operation.

The Attached Client Policy Set column can contain the following values:

- **None**. No policy set is attached, either directly or to a higher-level service resource.
- ***Policy_set_name***. The name of the policy set that is attached directly to the service resource, for example, WS-I RSP.

- ***Policy_set_name* (inherited)**. The name of the policy set that is not attached directly to a service resource, but that is attached to a higher-level service resource.

When the value in the column is a link, click the link to view or change settings about the attached policy set.

Binding

Specifies the binding configuration that is available for a service reference, endpoint, or operation.

The Binding column can contain the following values:

- **Not applicable**. No policy set is attached, either directly or to a higher-level service resource.
- ***Binding_name*** or **Default**. The binding name is displayed if a policy set is attached directly and an application-specific binding or a general binding is assigned, for example, `MyBindings1`. **Default** is displayed if a policy set is attached directly but the service resource uses the default bindings.
- ***Binding_name* (inherited)** or **Default (inherited)**. A service resource inherits the bindings from an attachment to a higher-level resource.

When the value in the Binding column is a link, click the link to view or change settings about the binding.

About policy set bindings

In this release, there are two types of bindings: application-specific bindings and general bindings. Composition units can use both application-specific bindings and general bindings.

Application-specific bindings

You can create application-specific bindings only at a policy set attachment point. These bindings are specific to, and constrained by, the characteristics of the defined policy. Application-specific bindings can provide configuration for advanced policy requirements such as multiple signatures; however, these bindings are reusable only within an application. Also, application-specific bindings have very limited reuse across policy sets.

When you create an application-specific binding for a policy set attachment, the binding begins in a completely unconfigured state. You must add each policy, such as WS-Security or HTTP transport, that you want to override the default binding, and fully configure the bindings for each policy that you add. For WS-Security policy, some high level configuration attributes such as TokenConsumer, TokenGenerator, SigningInfo, or EncryptionInfo might be obtained from the default bindings if they are not configured in the application-specific bindings.

For service providers, you can create application-specific bindings only by selecting **Assign Binding → New Application Specific Binding**, on the Service providers policy sets and bindings collection page, for service provider resources that have an attached policy set. Similarly, for service clients, you can create application-specific bindings only by selecting **Assign Binding → New Application Specific Binding**, on the Service clients policy sets and bindings collection page, for service client resources that have an attached policy set.

General bindings

You can configure general bindings to be used across a range of policy sets and they can be reused across applications and for trust service attachments. Although general bindings are highly reusable, they cannot provide configuration for advanced policy requirements such as multiple signatures. There are two types of general bindings: general provider policy set bindings and general client policy set bindings.

You can create general provider policy set bindings by clicking **Services** → **Policy sets** → **General provider policy set bindings** → **New** in the general provider policy sets panel, or by clicking **Services** → **Policy sets** → **General client policy set bindings** → **New** in the general client policy set and bindings panel. For details about defining and managing service client or provider bindings, see the related links. General provider policy set bindings might also be used for trust service attachments.

Example: Creating an SCA business-level application with the console

You can add many different types of artifacts to business-level applications. For example, you can add applications or modules, Java archives (JAR files), data in compressed files, and other business-level applications. This example describes how to create an empty business-level application and then add a Service Component Architecture (SCA) JAR file to the application using the administrative console.

Before you begin

Install the Feature Pack for SCA. Installing the feature pack adds SCA sample files to the *app_server_root*/installableApps directory. If you selected to install Samples during creation of a profile enabled by the feature pack, the product also adds several SCA sample files to the *app_server_root*/samples/SCA directory.

Also, verify that the target server is configured. As part of configuring the server, determine whether your application files can run on your deployment target. You must deploy SCA composite assets of a business-level application to a Version 7.0 server or cluster that is enabled for the Feature Pack for SCA.

About this task

For this example, use the administrative console to create a business-level application named HelloWorldAsync that has an SCA JAR file, helloworld-ws-asynch.jar, as an asset. The JAR file is available in *app_server_root*/installableApps.

1. Create an empty business-level application named HelloWorldAsync.
   a. Click **Applications** → **New Application** → **New Business Level Application**.
   b. On the New application page, specify the name HelloWorldAsync, optionally add a description, and then click **Apply**.
   c. On the page that is displayed, click the **Save** link.

   The name is shown in the list of applications on the Business-level applications page. Because the application is empty, its status is Unknown.
2. Import the SCA JAR asset.
   a. Click **Applications** → **New Application** → **New Asset** in the console navigation tree.
   b. On the Upload asset page, specify the asset package to import, helloworld-ws-asynch.jar, and click **Next**.

The JAR file is in the *app_server_root*/installableApps directory.

c. On the Select options for importing an asset page, click **Next** to accept the default values.

d. On the Summary page, click **Finish**.

e. On the Adding asset to repository page, if messages show that the operation completed, click **Manage assets**.

f. On the Assets page, click the **Save** link.

The file name displays in the list of assets.

3. Add the SCA JAR asset as a composition unit of the business-level application.

   a. Click **Applications** → **Application Types** → **Business-level applications**.

   b. On the Business-level applications page, click the HelloWorldAsync application name.

   c. On the business-level application settings page, click **Add** → **Add Asset**.

   d. On the Add page, select the helloworld-ws-asynch.jar asset composition unit from the list of available units, and then click **Continue**.

   e. On the Set options page, click **Next** to accept the default values.

   f. On the Map composition unit to a target page, specify a target server that is enabled for the Feature Pack for SCA, and then click **Next**.

      The target server can be an existing cluster. To map the composition unit to a cluster, select the existing cluster from the **Available** list, click **Add**, and then click **Next**. The cluster name is shown in the **Current targets** list as WebSphere:cluster=*cluster_name*.

   g. On the Define relationship with existing composition units page, click **Next** to accept the default values.

   h. On the Map virtual host page, click **Next** to accept the default values.

   i. On the Summary page, click **Finish**.

      Several messages are displayed. A message having the format Completed res=[WebSphere:cuname=helloworldws] indicates that the addition is successful.

      During deployment of the composition unit, you can view the Uniform Resource Identifier (URI) for composite level service of some bindings, along with the service name and binding type. Only the URI is editable. The product does not validate the URI.

   j. If the addition is successful, click **Manage application**.

   k. On the business-level application settings page, click **Save**.

   The asset name and type displays in the list of deployed assets. If you click on the asset name, the Composition unit settings page displays, with the asset name in the **SCA Composite Components** list.

4. Start the HelloWorldAsync business-level application.

   a. Click **Applications** → **Application Types** → **Business-level applications**.

   b. On the Business-level applications page, select the check box beside HelloWorldAsync.

   c. Click **Start**.

   When the business-level application is running, a green arrow displays for **Status**. If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again.

What to do next

Optionally examine, and possibly use in applications, other SCA sample files in *app_server_root*/installableApps or *app_server_root*/samples/SCA.

If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again. If SCA composite assets do not start, ensure that each asset is mapped to a deployment target that is enabled for the Feature Pack for SCA.

If an asset composition unit uses an Enterprise JavaBean (EJB) binding and does not start because it has a non-WebSphere target of "null", delete the asset composition unit and add it again to the business-level application. Specify a target enabled for the Feature Pack for SCA when you add the asset to the business-level application. You cannot change the target after deployment.

If the SCA application uses security, the target server or cluster must be in the global security domain.

In multiple-node environments, synchronize the nodes after you save changes to the target before starting the business-level application.

Chapter 10. Updating SCA composite artifacts

You can view and update Service Component Architecture (SCA) composite artifacts in business-level applications.

Before you begin

Add an SCA artifact as a composition unit to a business-level application.

About this task

You can view and update the following SCA composite artifacts:
- Composite level property definition
- Composite level component property definition
- Composite level component reference definition

You can view and update SCA composite artifacts using the administrative console or the wsadmin tool. This topic describes how to view and update SCA composite artifacts using the administrative console.

1. Go to the composition unit settings page for an SCA composite artifact in a business-level application.

   Click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***SCA_deployed_asset_composition_unit_name***.

   The composition unit settings page for an SCA composite artifact has fields that are not shown on the composition unit settings page for a non-SCA artifact:
   - SCA composite components
   - SCA composite properties
   - SCA composite wires
2. Click on a name link in one of these SCA fields to view the settings for an SCA artifact.

   The SCA fields display `None` instead of a name link if the composition unit does not have that particular type of SCA composite.
3. Optional: Update a SCA composite setting value.
   a. Change an existing setting value for the SCA artifact.
   b. Click **OK**.

   The setting value is updated.

Viewing and updating SCA composites in HelloWorldAsync

"Example: Creating an SCA business-level application with the console" on page 116 describes how to create the HelloWorldAsync business-level application. This application contains an SCA artifact, helloworldws, as a composition unit. You can view and update settings for SCA composites in the helloworldws composition unit using the console.

1. Go to the composition unit settings page for the helloworldws composition unit in the HelloWorldAsync business-level application.

   Click **Applications** → **Application Types** → **Business-level applications** → **HelloWorldAsync** → **helloworldws**.

   From the composition unit settings page, you can view information associated with helloworldws, as well as update composite settings.

2. Click on a link for the SCA artifact to be viewed or updated.

   For example, click on the HelloWorldServiceComponent link under **SCA composite components** and, in the page that displays, click on the HelloWorldService link under **Service**. In the Component service settings page that displays, you can specify a setting value for the service.
3. If you update a setting value for the SCA artifact, click **OK**.

What to do next

Save the changes to your administrative configuration.

On multiple-server products, when saving the configuration, synchronize the configuration with the nodes where the application is expected to run.

Chapter 11. Viewing SCA composite definitions

You can view information on the definition of a Service Component Architecture (SCA) composite in the administrative console.

Before you begin

The SCA composite must be a composition unit in a business-level application.

About this task

The composite definition provides data on the composite, such as component names and service references. The View composite page displays the composite definition of an SCA deployed asset composition unit.

1. Go to the View composite page.

   Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *SCA_deployed_asset_name* → **View composite**.
2. Optional: Click **Expand All** or **Collapse All** to more easily browse the page.

Results

The View composite page displays the contents of the composition unit definition.

Example

Suppose the HelloWorldAsync business-level application provided as a sample with the Feature Pack for SCA is installed. Click **Applications** → **Application Types** → **Business-level applications** → **HelloWorldAsync** → **helloworldws** → **View composite**.

The View composite page displays configuration information resembling the following:

```
<composite targetNamespace="http://helloworld" name="helloworldws" >
  <component name="AsynchTranslatorComponent" >
    <service name="HelloWorldService" >
      <implementation.java class="helloworld.impl.AsynchTranslatorComponent" />
      <service name="AsynchTranslatorService">
         <interface.java interface="helloworld.AsynchTranslatorService"
              callbackInterface="helloworld.HelloWorldCallback" />
        <binding.ws/>
          <callback>
              <binding.ws/>
          </callback>
      </service>
  </component>
</composite>
```

What to do next

Browse the page to ensure that it contains the intended configuration information.

Chapter 12. Viewing SCA domain information

You can view information on Service Component Architecture (SCA) composites in an SCA domain in the administrative console.

Before you begin

The SCA composite must be a composition unit in a business-level application.

About this task

Viewing SCA domain information enables you to see on one console page information on all components in an SCA domain. The View domain page displays information on available services in the current domain.

1. Go to the View domain page.

   Click **Applications** → **Application Types** → **Business-level applications** → *application_name* → *SCA_deployed_asset_name* → **View domain**.
2. Optional: Click **Expand All** or **Collapse All** to more easily browse the page.

Results

The View domain page lists information on components in the current domain.

Example

Suppose the HelloWorldAsync business-level application provided as a sample with the Feature Pack for SCA is installed. Click **Applications** → **Application Types** → **Business-level applications** → **HelloWorldAsync** → **helloworldws** → **View domain**.

The View domain page displays information resembling the following:

```
<domain name="myCell02">
  <component name = "HelloWorldServiceComponent"
        mapTarget = "WebSphere:cell=myCell02,node=myNode02,server=server1">
    <service name = "HelloWorldService">
      <interface.java interface = "helloworld.HelloWorldService"/>
    </service>
    <httpurlendpoints name = "endpoints" uri = ""/>
  </component>
</domain>
```

What to do next

Browse the page to ensure that it contains the intended information.

You can export the same domain information to a file using the exportCompositeToDomain command. See “Exporting SCA domain information using scripting.”

Chapter 13. Deleting business-level applications

After an application no longer is needed, you can delete it.

About this task

Deleting a business-level application removes the application from the product configuration repository and it deletes the application binaries from the file system of all nodes where the application files are installed.

1. Go to the Business-level applications page.

   Click **Applications** → **Application Types** → **Business-level applications** in the console navigation tree.
2. If you need to retain a copy of the application, back up composition units of the application.
3. Delete composition units of the application.
   a. On the Business-level applications page, click the name of the business-level application that you want to delete.
   b. On the business-level application settings page, delete each composition unit of the application. Deployed assets and business-level applications can be composition units of a business-level application.

      Select one or more composition units and click **Delete**.
   c. On the Delete composition unit from Business-level application panel, confirm the deletion and click **OK**.
   d. Repeat steps b and c until the business-level application that you want to delete has no more composition units.

   Deleting a composition unit removes the configuration from the *profile_root*/config/cells/*cell_name*/cus directory.
4. Delete the business-level application.
   a. Select the application that you want to delete.
   b. Click **Delete**.

   Unless the application is used by another business-level application, deleting a business-level application removes the configuration from the *profile_root*/config/cells/*cell_name*/blas directory.
5. On the Delete business-level application panel, confirm the deletion and click **OK**.
6. Save changes made to the administrative configuration.

Results

On single-server products, application binaries are deleted after you save the changes.

On multiple-server products, application binaries are deleted when configuration changes on the deployment manager synchronize with configurations for individual nodes.

Deleting the HelloWorldAsync business-level application

"Example: Creating an SCA business-level application with the console" on page 116 describes how to create the HelloWorldAsync business-level application. You can delete this application using the console.

1. Go to the business-level applications page and, if HelloWorldAsync is running, change its status to Stopped.
   a. Click **Applications** → **Application Types** → **Business-level applications**.
   b. Select HelloWorldAsync.
   c. Click **Stop**.
2. Go to the business-level applications settings page for HelloWorldAsync and delete the helloworldws composition unit.
   a. Click **Applications** → **Application Types** → **Business-level applications** → **HelloWorldAsync**.
   b. From **Deployed assets**, select helloworldws.
   c. Click **Delete**.
   d. On the Delete composition unit from Business-level application panel, confirm the deletion and click **OK**.
   e. Click the **Save** link to save the changes.
3. From the business-level applications page, delete the HelloWorldAsync application.
   a. Click **Applications** → **Application Types** → **Business-level applications**.
   b. Select HelloWorldAsync.
   c. Click **Delete**.
   d. On the Delete business-level application panel, click **OK**.
   e. Click the **Save** link to save the changes.
4. Optionally, from the Assets page, delete the helloworld-ws-asynch.jar asset from the asset repository.
   a. Click **Applications** → **Application Types** → **Assets**.
   b. Select helloworld-ws-asynch.jar.
   c. Click **Delete**.
   d. On the Delete asset panel, click **OK**.
   e. Click the **Save** link to save the changes.

What to do next

If using the administrative console **Delete** options does not fully delete a business-level application or its composition units, you can delete the business-level application and its composition units manually from a deployment manager or stand-alone server. Suppose you want to delete a business-level application named `ExampleBLA`, and `ExampleBLA` is not used by another business-level application. Complete the following steps to manually delete the `ExampleBLA` configurations from the `blas` and `cus` directories:

1. Delete the *profile_root*`/config/cells/`*`cell_name`*`/blas/ExampleBLA` directory.
2. Delete the *profile_root*`/config/cells/`*`cell_name`*`/cus/ExampleBLA` directory.
3. Save changes made to the administrative configuration.
4. On multiple-server products, synchronize the deployment manager with node configurations.

Chapter 14. Administering applications using wsadmin scripting

You can use administrative scripts and the wsadmin tool to install, uninstall, and manage applications.

About this task

There are two methods you can use to install, uninstall, and manage applications. You can use the commands for the AdminApp and AdminControl objects to invoke operations on your application configuration.

Alternatively, you can use the AdminApplication and BLAManagement Jython script libraries to perform specific operations to configure your enterprise and business-level applications.

The scripting library provides a set of procedures to automate the most common administration functions. You can run each script procedure individually, or combine several procedures to quickly develop new scripts.

You might need to complete one or more of the following topics to administer your application configurations with the wsadmin tool.

- Install enterprise applications. Use the AdminApp object or the AdminApplication script library to install an application to the application server runtime. You can install an enterprise archive file (EAR), Web archive (WAR) file, servlet archive (SAR), or Java archive (JAR) file.
- Install business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to install business-level applications.
- Manage enterprise applications using pattern matching. Use the AdminApp object or the AdminApplication script library to implement pattern matching when installing, updating, or editing an application. Pattern matching simplifies the task of supplying required values for certain complex options by allowing you to pass in asterisk (*) to all of the required values that cannot be edited.
- Manage Integrated Solutions Console applications. Use the AdminApp object to deploy or remove portlet-based Integration Solutions Console applications.
- Uninstall enterprise applications. Use the AdminApp object or the AdminApplication script library to uninstall applications.
- Uninstall business-level applications. Use the BLAManagement command group for the AdminTask object or the AdminBLA script library to uninstall business-level applications.
- Switch JavaServer Faces implementations. Use the modifyJSFImplementation command to set the Sun Reference Implementation or the Apache MyFaces project as the JSF implementation for Web applications.

Setting up business-level applications using wsadmin scripting

You can create an empty business-level application, and then add assets, shared libraries, or business-level applications as composition units to the empty business-level application.

Before you begin

Before you can create a business-level application, determine the assets or other files to add to your application.

Also, verify that the target application server is configured. As part of configuring the server, determine whether your application files can run on your deployment targets.

About this task

You can use the wsadmin tool to create business-level applications in your environment. This topic demonstrates how to use the AdminTask object to import and register assets, create empty business-level applications, and add assets to the business-level application as composition units. Alternatively, you can use the scripts in the AdminBLA script library to set up and administer business-level applications.

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Import assets to your configuration.

   Assets represent application binaries that contain business logic that runs on the target runtime environment and serves client requests. An asset can contain an archive of files such as a compressed (zip) or Java archive (JAR) file, or an archive of archive files such as a Java Platform, Enterprise Edition (Java EE) enterprise archive (EAR) file. Examples of assets include EAR files, shared library JAR files, and custom advisors for proxy servers.

   Use the importAsset command to import assets to the application server configuration repository. See the documentation for the BLAManagement command group for the AdminTask object for additional parameter and step options.

   For this example, the commands add three assets to the asset repository. Two of the assets are non-Java EE assets and one is an enterprise asset. The following command imports the `asset1.zip` asset to the asset repository and sets the returned configuration ID to the `asset1` variable:

```
asset1 = AdminTask.importAsset('-source c:/ears/asset1.zip')
```

```
asset1 = AdminTask.importAsset('-source \ears\asset1.zip')
```

   The following command imports the `asset2.zip` asset metadata only, sets the asset name as `testAsset.zip`, sets the deployment directory, specifies that the asset is used for testing, and sets the returned configuration ID to the `testasset` variable:

```
testasset = AdminTask.importAsset('-source c:/ears/asset2.zip -storageType
 METADATA –AssetOptions [[.* testAsset.zip .* "asset for testing"
 c:/installedAssets/testAsset.zip/BASE/testAsset.zip "" "" "" false]]')
```

```
testasset = AdminTask.importAsset('-source \ears\asset2.zip -storageType
 METADATA –AssetOptions [[.* testAsset.zip .* "asset for testing"
 c:/installedAssets/testAsset.zip/BASE/testAsset.zip "" "" "" false]]')
```

   The following command imports the `defaultapp.ear` asset, storing all application binaries, and sets the returned configuration ID to the `J2EEAsset` variable:

```
J2EEAsset = AdminTask.importAsset('-source c:/ears/defaultapplication.ear
 –storageType FULL –AssetOptions [[.* defaultapp.ear .* "desc" "" "" "" "" false]]')
```

```
J2EEAsset = AdminTask.importAsset('-source \ears\defaultapplication.ear
 –storageType FULL –AssetOptions [[.* defaultapp.ear .* "desc" "" "" "" "" false]]')
```

The assets of interest are registered as named configuration artifacts in the application server configuration repository, which is referred to as the asset registry. Use the listAssets command to display a list of registered assets and verify that the settings are correct, as the following example demonstrates:

```
AdminTask.listAssets('-includeDescription true -includeDeplUnit
 true')
```

3. Create an empty business-level application.

   Use the createEmptyBLA command to create a new business-level application and set the returned configuration ID to the `myBLA` variable, as the following example demonstrates:

```
myBLA = AdminTask.createEmptyBLA('-name myBLA -description "BLA that contains
 asset1, asset2, and J2EEAsset"')
```

   The system creates the business-level application. Use the listBLAs command to display a list of each business-level application in the cell, as the following example demonstrates:

```
AdminTask.listBLAs()
```

4. Add the assets, as composition units, to the business-level application.

   Composition units can represent deployed assets, other business-level applications, or external artifacts that are deployed on non-Application Server run times without backing assets. Business-level applications contain zero or more composition units. You cannot add the same composition unit to more than one business-level application, but you can use one asset to create more than one composition unit.

   The following command adds the `asset1.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` server:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset1 -CUOptions [[.* .*
 compositionUnit1 "composition unit that is backed by asset1" 0]] -MapTargets [[.* server1]]
 –ActivationPlanOptions [[.* specname=actplan0+specname=actplan1]]')
```

   The following command adds the `testAsset.zip` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('-blaID myBLA –cuSourceID asset2 -CUOptions [[.* .*
 compositionUnit2 "composition unit that is backed by asset2" 0]] -MapTargets [[.*
 server1+testServer]] –ActivationPlanOptions [.* specname=actplan0+specname=actplan1]')
```

   The following command adds the `defaultapp.ear` asset as a composition unit in the `myBLA` business-level application, and maps the deployment to the `server1` and `testServer` servers:

```
AdminTask.addCompUnit('[-blaID bla1 -cuSourceID ' + J2EEAsset + '
 -defaultBindingOptions
 defaultbinding.ejbjndi.prefix=ejb#defaultbinding.virtual.host=default_host#defaultbinding.force=yes
-AppDeploymentOptions [-appname defaultapp] -MapModulesToServers [["Default Web Application" .*
 WebSphere:cell=cellName,node=nodeName,server=server1] ["Increment EJB module" .*
 Websphere:cell=cellName,node=nodeName,server=testServer]] -CtxRootForWebMod [["Default Web Application" .*
 myctx/]]]')
```

5. Save your configuration changes.

   Use the following command example to save your configuration changes:

```
AdminConfig.save()
```

6. Synchronize the nodes.

   Use the syncActiveNodes script in the AdminNodeManagement script library to synchronize each active node in your environment, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

7. Start the business-level application.

Use the startBLA command to start each composition unit of the business-level application on the deployment targets for which the composition units are configured, as the following example demonstrates:

```
AdminTask.startBLA('-blaID myBLA')
```

Results

The system adds three composition units backed by assets to a new business-level application. Each of the three assets are deployed and started on the `server1` server. The `testAsset.zip` and `defaultapp.ear` assets are also deployed and started on the `testServer` server.

Example: Creating an SCA business-level application with scripting

You can add many different types of artifacts to business-level applications. For example, you can add applications or modules, Java archives (JAR files), data in compressed files, and other business-level applications. This example describes how to create an empty business-level application and then add a Service Component Architecture (SCA) JAR file to the application using scripting.

Before you begin

Install the Feature Pack for SCA. Installing the feature pack adds SCA sample files to the *app_server_root*`/installableApps` directory. If you selected to install Samples during creation of a profile enabled by the feature pack, the product also adds several SCA sample files to the *app_server_root*`/samples/SCA` directory.

Also, verify that the target server is configured. As part of configuring the server, determine whether your application files can run on your deployment target. You must deploy SCA composite assets of a business-level application to a Version 7.0 server or cluster that is enabled for the Feature Pack for SCA.

About this task

For this example, use wsadmin scripts in the Jython or Jacl language to create a business-level application named HelloWorldAsync that has an SCA JAR file, `helloworld-ws-asynch.jar`, as an asset. The JAR file is available in *app_server_root*`/installableApps`.

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Create an empty business-level application named `HelloWorldAsync`.

   Use the createEmptyBLA command to create the business-level application.

   Using Jython:

   ```
   AdminTask.createEmptyBLA('-name HelloWorldAsync')
   ```

   Using Jacl:

   ```
   $AdminTask createEmptyBLA {-name HelloWorldAsync}
   ```

   After the command runs, the blaID output displays in the command window:

   ```
   WebSphere:blaname=HelloWorldAsync
   ```

   You can run the listBLAs command to view a list of all business-level applications in the cell and to confirm that the HelloWorldAsync business-level application exists.

   Using Jython:

   ```
   AdminTask.listBLAs()
   ```

To view a more readable list of business-level applications, try `print` before the command:

```
print AdminTask.listBLAs()
```

Using Jacl:

```
$AdminTask listBLAs
```

3. Import the SCA JAR asset.

   Use the importAsset command to import the JAR file to the product configuration repository.

   Using Jython:

   ```
   AdminTask.importAsset('-source app_server_root/installableApps/helloworld-ws-asynch.jar')
   ```

   Using Jacl:

   ```
   $AdminTask importAsset {-source app_server_root/installableApps/helloworld-ws-asynch.jar}
   ```

   After the command runs, the assetID output displays in the command window:

   ```
   WebSphere:assetname=helloworld-ws-asynch.jar
   ```

   You can run the listAssets command to view a list of all assets in the cell and to confirm that the helloworld-ws-asynch.jar asset exists:

   Using Jython:

   ```
   AdminTask.listAssets()
   ```

   To view a more readable list of assets, try `print` before the listAssets command:

   ```
   print AdminTask.listAssets()
   ```

   Using Jacl:

   ```
   $AdminTask listAssets
   ```

4. Add the SCA JAR asset as a composition unit of the business-level application.

   Use the addCompUnit command to add the asset to the business-level application.

   Using Jython:

   ```
   AdminTask.addCompUnit('[-blaID HelloWorldAsync -cuSourceID helloworld-ws-asynch.jar
     -MapTargets [[ .* SCA_server_name ]] ]')
   ```

   Using Jacl:

   ```
   $AdminTask addCompUnit {-blaID HelloWorldAsync -cuSourceID helloworld-ws-asynch.jar
     -MapTargets {{ .* SCA_server_name }} }
   ```

   SCA_server_name is the name of the target server or cluster; for example, `server1`. The target must be enabled for the Feature Pack for SCA.

   After the command runs, the composition unit ID output displays in the command window:

   ```
   WebSphere:cuname=helloworldws
   ```

   During deployment of the composition unit, you can view the Uniform Resource Identifier (URI) for composite level service of some bindings, along with the service name and binding type. Only the URI is editable. The product does not validate the URI.

   You can run the listCompUnits command to view a list of all composition units in a specified business-level application and to confirm that the helloworldws composition unit exists in HelloWorldAsync.

   Using Jython:

   ```
   AdminTask.listCompUnits('-blaID HelloWorldAsync')
   ```

   Using Jacl:

   ```
   $AdminTask listCompUnits {-blaID HelloWorldAsync}
   ```

5. Save the configuration changes.

Using Jython:

```
AdminConfig.save()
```

Using Jacl:

```
$AdminConfig save
```

6. Start the HelloWorldAsync business-level application.

   Use the startBLA command to start the application.

   Using Jython:

   ```
   AdminTask.startBLA('-blaID HelloWorldAsync')
   ```

   Using Jacl:

   ```
   $AdminTask startBLA {-blaID HelloWorldAsync}
   ```

   Verify that you see the following message indicating that the application started successfully:

   ```
   CWWMH0196I: BLA "WebSphere:blaname=HelloWorldAsync" was started successfully.
   ```

   Optionally, query the status to see whether the application is running with the getBLAStatus command.

   Using Jython:

   ```
   AdminTask.getBLAStatus('-blaID HelloWorldAsync')
   ```

   Using Jacl:

   ```
   $AdminTask getBLAStatus {-blaID HelloWorldAsync}
   ```

   The following message indicates that the application is started:

   ```
   BLA: WebSphere:blaname=HelloWorldAsync State of BLA WebSphere:blaname=HelloWorldAsync
     is Started.
   ```

7. Exit the wsadmin command shell.

   ```
   exit
   ```

What to do next

Optionally examine, and possibly use in applications, other SCA sample files in *app_server_root*/installableApps or *app_server_root*/samples/SCA.

If the business-level application does not start, ensure that the deployment target to which the application maps is running and try starting the application again. If SCA composite assets do not start, ensure that each asset is mapped to a deployment target that is enabled for the Feature Pack for SCA.

If an asset composition unit uses an Enterprise JavaBean (EJB) binding and does not start because it has a non-WebSphere target of "null", delete the asset composition unit and add it again to the business-level application. Specify a target enabled for the Feature Pack for SCA when you add the asset to the business-level application. You cannot change the target after deployment.

If the SCA application uses security, the target server or cluster must be in the global security domain.

In multiple-node environments, synchronize the nodes after you save changes to the target before starting the business-level application.

Deleting business-level applications using wsadmin scripting

You can use the wsadmin tool to remove business-level applications from your environment. Deleting a business-level application removes the application from the product configuration repository and it deletes the application binaries from the file system of all nodes where the application files are installed.

Before you begin

There are two ways to complete this task. This topic uses the commands in the BLAManagement command group for the AdminTask object to remove business-level applications from your configuration. Alternatively, you can use the scripts in the AdminBLA script library to configure, administer, and remove business-level applications

About this task

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Verify that the business-level application is ready to be deleted.

   Before deleting a business-level application, use the deleteCompUnit command to remove each configuration unit that is associated with the business-level application. Also, verify that no other business-level applications reference the business-level application to delete.

   Use the following example to delete the composition units for the business-level application of interest:

```
AdminTask.deleteCompUnit('-blaID myBLA –cuID compositionUnit1')
```

   Repeat this step for each composition unit that is associated with the business-level application of interest.
3. Delete the business-level application.

   Use the deleteBLA command to remove a business-level application from your configuration, as the following example demonstrates:

```
AdminTask.deleteBLA('-blaID myBLA')
```

   If the system successfully deletes the business-level application, the command returns the configuration ID of the deleted business-level application, as the following example displays:

```
WebSphere:blaname=myBLA
```

4. Save your configuration changes.

   Use the following command example to save your configuration changes:

   ```
   AdminConfig.save()
   ```
5. Synchronize the node.

   Use the syncActiveNodes script in the AdminNodeManagement script library to propagate the changes to each active node, as the following example demonstrates:

```
AdminNodeManagement.syncActiveNodes()
```

Deleting the HelloWorldAsync business-level application

"Example: Creating an SCA business-level application with scripting" on page 130 describes how to create the HelloWorldAsync business-level application. You can delete this application using wsadmin commands in the Jython scripting language.

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Stop the HelloWorldAsync business-level application.

   ```
   AdminTask.stopBLA('-blaID WebSphere:appName=HelloWorldAsync')
   ```

3. Delete the helloworldws composition unit associated with the HelloWorldAsync.
   ```
   AdminTask.deleteCompUnit('-blaID HelloWorldAsync -cuID helloworldws')
   ```
4. Delete the HelloWorldAsync application.
   ```
   AdminTask.deleteBLA('-blaID HelloWorldAsync')
   ```
5. Optionally, delete the helloworld-ws-asynch.jar asset from the asset repository.
   ```
   AdminTask.deleteAsset('-assetID helloworld-ws-asynch.jar')
   ```
6. Save the configuration changes.
   ```
   AdminConfig.save()
   ```
7. Exit the wsadmin command shell.
   ```
   exit
   ```

Chapter 15. Managing deployed applications using wsadmin scripting

Use these topics to learn more about managing deployed applications with the wsadmin tool and scripting.

- Start enterprise applications and stop enterprise applications. You can use the wsadmin tool and the AdminControl object to start an application that is not running (has a status of Stopped) or stop an application that is running (has a status of Started).
- Start business-level applications and stop business-level applications. You can use the wsadmin tool and the BLAManagement command group to start and stop business-level applications.
- Update applications. Use the wsadmin tool to update installed applications on an application server.
- Manage assets. Use the wsadmin tool and commands in the BLAManagement command group to manage your asset configuration. This topic provides examples for listing assets, viewing asset configuration data, removing assets from the asset repository, updating one or more files for assets, and exporting assets.
- Manage composition units. Use the wsadmin tool and commands in the BLAManagement command group to manage composition units. This topic provides examples for adding, removing, editing, exporting, and viewing composition units.
- List application modules. Use the wsadmin tool and the AdminApp object listModules command to list the modules in an installed application.
- Query the application state. Use the wsadmin tool and scripting to determine if an application is running.
- Disable application loading. You can use the wsadmin tool and the AdminConfig object to disable application loading in deployed targets.
- Export applications. You can use the wsadmin tool and the AdminApp object to export your applications.

Exporting SCA domain information using scripting

You can export information on Service Component Architecture (SCA) composites in an SCA domain to a file of your choice.

Before you begin

An SCA composite must be a composition unit in a business-level application.

About this task

Note: You can view information on components in an SCA domain. The SCA domain is typically the cell on multiple-server installations and the server scope on single-server installations. You can view SCA domain information in the administrative console or by exporting it to a file using scripting. Exporting SCA domain information enables you to preserve information on components.

This topic describes how to export domain information using scripting.

You might export domain information before updating SCA business-level applications or before migrating to a later version of the product.

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Optional: View online help for the exportCompositeToDomain command.

   Using Jython:

   ```
   print AdminTask.help('exportCompositeToDomain')
   ```

   Using Jacl:

   ```
   $AdminTask help exportCompositeToDomain
   ```
3. Export information on SCA composites in a domain to a file of your choice.

   Use the exportCompositeToDomain command to export the information. The command has two parameters, -domainName and -fileName, both of type String. The -domainName parameter is optional. The -fileName parameter is required.

   Using Jython:

   ```
   AdminTask.exportCompositeToDomain('[-domainName SCA_domain_name -fileName C:/my_file]')
   ```

   ```
   AdminTask.exportCompositeToDomain('[-domainName SCA_domain_name -fileName /my_file]')
   ```

   Using Jacl:

   ```
   $AdminTask exportCompositeToDomain {-domainName SCA_domain_name -fileName C:/my_file}
   ```

   ```
   $AdminTask exportCompositeToDomain {-domainName SCA_domain_name -fileName /my_file}
   ```

Table 38. exportCompositeToDomain command elements

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object to run administrative commands with the wsadmin tool
`exportCompositeToDomain`	is an AdminTask command
SCA_domain_name	is the name of SCA domain whose information is exported
/my_file	is the name of the file to which domain information is written

Results

After the exportCompositeToDomain command runs, information on components in the SCA domain is written to the specified file. The product displays the following message:

SCA_domain_name `exported to` */my_file*`.`

You can view the same domain information in the administrative console. Click **Applications** → **Application Types** → **Business-level applications** → ***application_name*** → ***deployed_asset_name*** → **View domain**.

Example

Suppose the HelloWorldAsync business-level application provided as a sample with the Feature Pack for SCA is installed. Run the exportCompositeToDomain command to export the composites:

Using Jython:

```
AdminTask.exportCompositeToDomain('[ -fileName C:/my_file ]')
```

```
AdminTask.exportCompositeToDomain('[ -fileName /my_file ]')
```

Using Jacl:

```
$AdminTask exportCompositeToDomain { -fileName C:/my_file }
```

```
$AdminTask exportCompositeToDomain { -fileName /my_file }
```

Running the exportCompositeToDomain command writes domain information resembling the following to the specified file:

```
<?xml version="1.0" encoding="UTF-8"?>
<domain name="myDomain">
<component name = "HelloWorldServiceComponent"
           mapTarget = "WebSphere:cell=myCell02,node=myNode02,server=server1">
<service name = "HelloWorldService">
<interface.java interface = "helloworld.HelloWorldService"/>
</service>
<httpurlendpoints name = "endpoints" uri = ""/>
</component>
</domain>
```

What to do next

Examine the exported file to ensure that it contains the intended information.

Exporting WSDL and XSD documents using scripting

You can export Web Services Description Language (WSDL) and XML schema definition (XSD) documents used by a Service Component Architecture (SCA) composition unit to a location of your choice.

Before you begin

Your SCA business-level application must contain one or more composition units that use a WSDL or XSD document.

A WSDL document is a file that provides a set of definitions that describe a Web service in WSDL, an Extensible Markup Language (XML)-based description language.

An XSD document is an instance of an XML schema written in the XML schema definition language. The document has the extension `.xsd`. The prefix `xsd` in the XML elements of an XSD document indicates the XML schema namespace.

About this task

Note: You can export WSDL and XSD documents used by an SCA composition unit using the exportWSDLArtifacts command.

Run the exportWSDLArtifacts command to extract from a specified composition unit the WSDL and XSD files that are required for Web services client development. The command extracts files for the services exposed by the Web service binding, `binding.ws`.

1. Start the wsadmin scripting tool using the Jython scripting language.
2. Optional: View online help for the exportWSDLArtifacts command.

   Using Jython:

   ```
   AdminTask.help('exportWSDLArtifacts')
   ```

   Using Jacl:

   ```
   $AdminTask help exportWSDLArtifacts
   ```
3. Export the WSDL and XSD documents to a location of your choice.

   Use the exportWSDLArtifacts command to export the WSDL and XSD documents. The command has two required parameters, -cuName and -exportDir, both of type String.

   Using Jython:

   ```
   AdminTask.exportWSDLArtifacts('[-cuName composition_unit_name -exportDir C:/my_directory]')
   ```

   ```
   AdminTask.exportWSDLArtifacts('[-cuName composition_unit_name -exportDir /my_directory]')
   ```

   Using Jacl:

   ```
   $AdminTask exportWSDLArtifacts {-cuName composition_unit_name -exportDir C:/my_directory}
   ```

   ```
   $AdminTask exportWSDLArtifacts {-cuName composition_unit_name -exportDir /my_directory}
   ```

Table 39. exportWSDLArtifacts command elements

`$`	is a Jacl operator for substituting a variable name with its value
`AdminTask`	is an object to run administrative commands with the wsadmin tool
`exportWSDLArtifacts`	is an AdminTask command
composition_unit_name	is the name of the composition unit whose WSDL or XSD documents are exported
/my_directory	is the absolute path of the directory to which the WSDL or XSD documents are exported

Results

After the exportWSDLArtifacts command runs, the following message displays in the command window:

```
'CWSAM0503I: WSDL Artifacts have been exported successfully.'
```

Example

Suppose you want to export WSDL or XSD documents in the HelloWorldAsync business-level application provided as a sample with the Feature Pack for SCA. Run the following command in the Jython scripting language to export documents in the helloworldws composition unit:

```
AdminTask.exportWSDLArtifacts('[-cuName helloworldws -exportDir C:/my_directory]')
```

```
AdminTask.exportWSDLArtifacts('[-cuName helloworldws -exportDir /my_directory]')
```

To run the command, the `my_directory` directory must exist on the computer.

Running the exportWSDLArtifacts command adds the `helloworldws_WSDLArtifacts.zip` file to the specified directory. The `helloworldws_WSDLArtifacts.zip` file has two WSDL files, one in the main file directory and one in the `/WEB-INF/wsdl/` subdirectory:

```
HelloWorldService_wsdlgen.wsdl
/WEB-INF/wsdl/helloworld.wsdl
```

What to do next

Examine the exported files to ensure that they contain the intended WSDL and XSD documents.

Chapter 16. Authorizing access to resources

WebSphere Application Server provides many different methods for authorizing accessing resources. For example, you can assign roles to users and configure a built-in or external authorization provider.

About this task

You can create an application, an Enterprise JavaBeans (EJB) module, or a Web module and secure them using assembly tools.

To authorize user or group access to resources, read the following articles:

1. Secure you application during assembly and deployment. For more information on how to create a secure application using an assembly tool, such as the IBM Rational® Application Developer, see the information about securing applications during assembly and deployment.
2. Authorize access to Java Platform, Enterprise Edition (Java EE) resources. WebSphere Application Server supports authorization that is based on the Java Authorization Contract for Containers (JACC) specification in addition to the default authorization. When security is enabled in WebSphere Application Server, the default authorization is used unless a JACC provider is specified.
3. Authorize access to administrative resources. You can assign users and groups to predefined administrative roles such as the monitor, configurator, operator, administrator, auditor and iscadmins roles. These roles determine which tasks a user can perform in the administrative console.

What to do next

After authorizing access to resources, configure the Application Server for secure communication. .

Using SCA authorization and security identity policies

Use two Service Component Architecture (SCA) declarative policies (*authorization* and *security identity*) to protect SCA components and operations and to declare the security identity under which the SCA components or operations are executed.

Before you begin

A user registry must be configured and an SCA component must first have been developed. You must also enable application security.

About this task

Note: An authorization policy controls who can access protected SCA components and operations. A security identity policy declares the security identity under which an SCA component or operation is executed. You can limit access to an SCA component or to an operation to particular users or groups, You can also delegate access to another user when executing an SCA component or an operation.

Note the following limitations:

- SCA authorization policy is not supported for composites packaged in Web application archives (WAR files).
- The `definitions.xml` file must be packaged in the same asset as the composites that reference its policy sets.
- Role assignments are scoped to a configuration unit, and are required for all of the roles used in all of the composites within the configuration unit. These role assignments are completely independent of any role assignments made for other configuration units in the same business-level application.
- The target namespace of the policy set and the name of the policy set do not contribute to the name of a role. They are used solely to resolve the policy set reference. This implies that within the same configuration unit, identically-named roles that are defined within different policy sets or different name spaces are treated as the same role.
- If authorization policy is not attached to a given component and operation, the operation runs unprotected.
- It is possible to create conflicts by specifying multiple policy sets in the @policySets attribute or by inheriting policy sets across elements. In this case, the following rules are used:
  - The **<denyAll>** element takes precedence over **<permitAll>**, which takes precedence over **<allow>**.
  - Roles from multiple **<allow>** elements are aggregated.
- SCA authorization policy does not support authorizing users in foreign realms.

Access to an SCA component is permitted or denied by using the following steps:

1. The policy administrator creates one or more policy sets in the file named `definitions.xml` as shown in the following example:

```
<definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
  targetNamespace="http://smallvilleBank"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0">
  <policySet name="StaffAuthorizationPolicy"
  appliesTo="sca:implementation"
  xmlns="http://www.osoa.org/xmlns/sca/1.0">
    <authorization>
      <allow roles="staff"/>
    </authorization>
  </policySet>
  <policySet name="SupervisorAuthorizationPolicy"
  appliesTo="sca:implementation"
  xmlns="http://www.osoa.org/xmlns/sca/1.0">
    <authorization>
      <allow roles="supervisor manager specialist"/>
    </authorization>
    <securityIdentity>
      <runAs role="specialist"/>
    </securityIdentity>
 </policySet>
</definitions>
```

2. The assembler attaches the policy to the SCA composite as in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:bank="http://smallvilleBank"
  name="AccountServices">
  <component name="AccountAccess">
        <implementation.java class="smallvilleBank.AccountAccessImpl"
             policySets="bank:staffAuthorizationPolicy"/>
  </component>
  <component name="AccountAudit">
```

```
            <implementation.java class="smallvilleBank.AccountAuditImpl"
                policySets="bank:supervisorAuthorizationPolicy"/>
        </component>
    </composite>
```

3. The deployer assigns users and or groups to the roles that are defined in the composite.
4. The deployer assigns a user to the **runAs** roles that are defined in the composite.

What to do next

Access to the SCA component is permitted or denied according to the authorization policy.

Using the SCA RequestContext.getSecuritySubject() API

The Service Component Architecture (SCA) Version 1.0 Java Common Annotations and APIs Specification RequestContext.getSecuritySubject() API programming interface returns a Java Authentication and Authorization (JAAS) subject that represents an authenticated user who accesses the protected SCA service.

Before you begin

Note: SCA service developers can use the RequestContext.getSecuritySubject() API to obtain a JAAS Subject that represents the requester.

If one or more of the following preconditions are not met the SCA request is not authenticated, and the RequestContext.getSecuritySubject API returns a null Subject:

- Administrative security must be enabled to initialize the security infrastructure.
- Application security must be enabled to enforce security policy and authentication.
- SCA service either has an authentication intent or a PolicySet that requires authentication prior to deployment.
- The SCA deployment process must associate a PolicySet that contains authentication policy configuration to the SCA service.

About this task

When using the RequestContext.getSecuritySubject() API, perform the following steps:

1. Add an authentication intent or specify a PolicySet in the binding element of an SCA service composite file to enforce SCA request authentication, as shown in the following example. The following example uses the ″authentication.transport″ intent.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           xmlns:dbsdo="http://tuscany.apache.org/xmlns/sca/databinding/sdo/1.0"
           xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
           xmlns:qos="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           name="EchoServiceWithIdentityWSComposite">
    <component name="EchoServiceWithIdentityWSComponent">
        <implementation.java class="test.ws.soa.sca.qos.policy.echoRelayServiceTest.echoService.EchoServiceWithIdentityComponentImpl"/>
        <service name="EchoService">
            <binding.ws uri="EchoServiceWithIdentity"
                wsdlElement="http://echo#wsdl.port(EchoServiceWithIdentity/EchoServiceWithIdentitySoapPort)"
                requires="authentication.transport" />
        </service>
    </component>
</composite>
```

2. Specify the ″WSHTTPS default″ PolicySet in the SCA client composite file. A user name and password are configured for use in outbound requests of the ″HTTP Transport″ default PolicySet binding.

   The following example utilizes the RequestContext.getSecuritySubject API:

```
import org.osoa.sca.CompositeContext;
import org.osoa.sca.CurrentCompositeContext;
import org.osoa.sca.RequestContext;
import javax.security.auth.Subject;
import java.security.Principal;
import java.util.Iterator;
import com.ibm.websphere.security.cred.WSCredential;

. . .
    try {
         CompositeContext compositeContext = CurrentCompositeContext.getContext();
    RequestContext requestContext = null;
    Subject subject = null;
    String securityName = null;
          if (compositeContext != null) {
               requestContext = compositeContext.getRequestContext();
    }
          if (requestContext != null) {
               subject = requestContext.getSecuritySubject();
    }
          if (subject != null) {
               java.util.Set principalSet = subject.getPrincipals();
              if (principalSet != null && principalSet.size() > 0) {
                     Iterator principalIterator = principalSet.iterator();
                     if (principalIterator.hasNext()) {
                         Principal principal = (java.security.Principal) principalIterator.next();
securityName = principal.getName();
                     }
               }
   }
```

3. The principal identity consists of a realm name followed by the identity of the requester as shown in the example below. WebSphere Application Server is configured to use an Lightweight Directory Access Protocol (LDAP) server for authentication. The realm name is the LDAP server host name and the port number:

   ```
   security name = ldap1.austin.ibm.com:389/user2
   ```

   You can obtain various security attributes of the request from the WSCredential object in the subject as shown in the following example:

```
if (subject != null) {
    java.util.Set credSet = subject.getPublicCredentials();
    if (credSet != null && credSet.size() > 0)
    {
        Iterator credIterator = credSet.iterator();
        while (credIterator.hasNext()) {
            Object o = credIterator.next();
            WSCredential cred = null;
            if (o instanceof WSCredential) {
                cred = (WSCredential) o;
            } else {
                if (securityName == null) {
                    securityName = new StringBuffer();
                }
                securityName.append("\n>> Found a public credential: " + o.getClass().getName());
            }
            if (cred != null) {
                if (securityName == null) {
                    securityName = new StringBuffer();
                }
                securityName.append("\n>> WSCredential security attributes . . .");
                securityName.append("\n>> getAccessId = \t\t" + cred.getAccessId());
                securityName.append("\n>> getGroupIds = \t\t" + cred.getGroupIds());
                securityName.append("\n>> getPrimaryGroupId = \t\t" + cred.getPrimaryGroupId());
                securityName.append("\n>> getRealmName = \t\t" + cred.getRealmName());
                securityName.append("\n>> getRealmSecurityName = \t\t" + cred.getRealmSecurityName());
                securityName.append("\n>> getRealmUniqueSecurityName = \t\t" + cred.getRealmUniqueSecurityName());
                securityName.append("\n>> getSecurityName = \t\t" + cred.getSecurityName());
                securityName.append("\n>> getUniqueSecurityName = \t\t" + cred.getUniqueSecurityName());
            }
        }
    }
}
```

Sample output is shown below:

```
>> WSCredential security attributes . . .
>> getAccessId =    user:ldap1.austin.ibm.com:389/cn=user2,o=ibm,c=us
>> getGroupIds =    [group:ldap1.austin.ibm.com:389/CN=GROUP2,O=IBM,C=US]
```

```
>> getPrimaryGroupId =   group:ldap1.austin.ibm.com:389/CN=GROUP2,O=IBM,C=US
>> getRealmName =   ldap1.austin.ibm.com:389
>> getRealmSecurityName =  ldap1.austin.ibm.com:389/user2
>> getRealmUniqueSecurityName = ldap1.austin.ibm.com:389/cn=user2,o=ibm,c=us
>> getSecurityName =   user2
>> getUniqueSecurityName =  cn=user2,o=ibm,c=us
```

Chapter 17. Using JAXB for XML data binding

Java Architecture for XML Binding (JAXB) is a Java technology that provides an easy and convenient way to map Java classes and XML schema for simplified Web services development. JAXB provides the xjc schema compiler, the schemagen schema generator and a runtime framework to support marshalling and unmarshalling of XML documents to and from Java objects.

About this task

JAXB is an XML-to-Java binding technology that enables transformation between schema and Java objects and between XML instance documents and Java object instances. JAXB technology consists of a runtime API and accompanying tools that simplify access to XML documents. You can use JAXB APIs and tools to establish mappings between Java classes and XML schema. An XML schema defines the data elements and structure of an XML document. JAXB technology provides tooling to enable you to convert your XML documents to and from Java objects. Data stored in an XML document is accessible without the need to understand the XML data structure.

JAXB is the default data binding technology used by the Java API for XML Web Services (JAX-WS) tooling and implementation within this product. You can develop JAXB objects to use within your JAX-WS applications. You can also use JAXB independently of the JAX-WS programming model as a convenient way to leverage the XML data binding technology to manipulate XML within your Java applications.

JAXB is also the default data binding technology used by Service Component Architecture (SCA) applications. JAXB enables the SCA service implementation side and the SCA client reference side to interact with Java objects without worrying about how the data is transformed into and from XML. JAXB is supported for the binding.sca and binding.ws binding types.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

JAXB provides the xjc schema compiler tool, the schemagen schema generator tool, and a runtime framework. The xjc schema compiler tool enables you to start with an XML schema definition (XSD) to create a set of JavaBeans that map to the elements and types defined in the XSD schema. You can also start with a set of JavaBeans and use the schemagen schema generator tool to create the XML schema. After using either the schema compiler or the schema generator command-line tools, you can convert your XML documents both to and from Java objects and use the resulting Java classes to assemble a Web services application.

In addition to using the tools from the command-line, you can invoke these JAXB tools from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask`

Ant task from within the Ant build environment to invoke the xjc schema compiler tool. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

JAXB annotated classes and artifacts contain all the information that the JAXB runtime API needs to process XML instance documents. The JAXB runtime API enables marshaling of JAXB objects to XML files and unmarshaling the XML document back to JAXB class instances. The JAXB binding package, `javax.xml.bind`, defines the abstract classes and interfaces that are used directly with content classes. In addition the package defines the marshal and unmarshal APIs.

JAXB 2.1 provides enhancements such as improved compilation support and support for the @XMLSeeAlso annotation. With JAXB 2.1, you can configure the xjc schema compiler so that it does not automatically generate new classes for a particular schema. Similarly, you can configure the schemagen schema generator to not automatically generate a new schema. This enhancement is useful when you are using a common schema and you do not want a new schema generated. JAXB 2.1 also introduces the @XMLSeeAlso annotation that enables JAXB to bind additional Java classes that it might not otherwise know about when binding a Java class with this annotation. This annotation enables JAXB to know about all classes that are potentially involved in marshalling or unmarshalling as it is not always possible or practical to list all of the subclasses of a given Java class. JAX-WS 2.1 also supports the use of the @XMLSeeAlso annotation on a service endpoint interface (SEI) or on a service implementation bean to ensure all of the classes referenced by the annotation are passed to JAXB for processing.

You can optionally use JAXB binding customizations to customize generated JAXB classes by overriding or extending the default JAXB bindings when the default bindings do not meet your business application needs. In most cases, the default binding rules are sufficient to generate a robust set of schema-derived classes. JAXB supports binding customizations and overrides to the default binding rules that you can make through various ways. For example, you can the overrides inline as annotations in a source schema, as declarations in an external bindings customization file that is used by the JAXB binding compiler, or as Java annotations within Java class files used by the JAXB schema generator. See the JAXB specification for information regarding binding customization options.

Using JAXB, you can manipulate data objects in the following ways:

- Generate an XML schema from a Java class. Use the schema generator schemagen command to generate an XML schema from Java classes.
- Generate Java classes from an XML schema. Use the schema compiler xjc command to create a set of JAXB-annotated Java classes from an XML schema.
- Marshal and unmarshal XML documents. After the mapping between XML schema and Java classes exists, use the JAXB binding runtime to convert XML instance documents to and from Java objects.

Results

You now have JAXB objects that your Java application can use to manipulate XML data.

Using JAXB schemagen tooling to generate an XML schema file from a Java class

Use Java Architecture for XML Binding (JAXB) schemagen tooling to generate an XML schema file from Java classes.

Before you begin

Identify the Java classes or a set of Java objects to map to an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between Java classes and XML schema. XML schema documents describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a bottom-up development approach starting from existing JavaBeans or enterprise beans, use the wsgen tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications or the Service Component Architecture (SCA) representations of your business service interfaces. After the Java artifacts for your application are generated, you can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

You can create an XML schema document from an existing Java application that represents the data elements of a Java application by using the JAXB schema generator, schemagen command-line tool. The JAXB schema generator processes either Java source files or class files. Java class annotations provide the capability to customize the default mappings from existing Java classes to the generated schema components. The XML schema file along with the annotated Java class files contain all the necessary information that the JAXB runtime requires to parse the XML documents for marshaling and unmarshaling.

Note: The wsimport, wsgen, schemagen and xjc command-line tools are not supported on the z/OS platform. This functionality is provided by the assembly tools provided with WebSphere Application Server running on the z/OS platform. Read about these command-line tools for JAX-WS applications to learn more about these tools.

Note: WebSphere provides Java API for XML-Based Web Services (JAX-WS) and Java Architecture for XML Binding (JAXB) tooling. The wsimport, wsgen, schemagen and xjc command-line tools are located in the *app_server_root*\bin\ directory. Similar tooling is provided by the Java SE Development Kit (JDK) 6. For the most part, artifacts generated by both the tooling provided with WebSphere and the JDK are the same. In general, artifacts generated by the JDK tools are portable across compliant runtime

environments. However, it is a best practice to use the WebSphere tools to achieve seamless integration within the WebSphere environment.

Note: WebSphere Application Server Version 7.0 supports the JAXB 2.1 specification. JAX-WS 2.1 requires JAXB 2.1 for data binding.

JAXB 2.1 provides improvements in compilation support to enable you to configure the schemagen schema generator so that it does not automatically generate a new schema. This is helpful if you are using a common schema such as the World Wide Web Consortium (W3C), XML Schema, Web Services Description Language (WSDL), or WS-Addressing and you do not want a new schema generated for a particular package that is referenced. The location attribute on the @XmlSchema annotation causes the schemagen generator to refer to the URI of the existing schema instead of generating a new one.

In addition to using the schemagen tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.jxc.SchemaGenTask` Ant task from within the Ant build environment to invoke the schemagen schema generator tool.

Note: When running the schemagen tool, the schema generator does not correctly read the @XmlSchema annotations from the package-info class file to derive targetNamespaces. Instead of using the @XMLSchema annotation, use one of the following methods:

- Provide a package-info.java file with the @XmlSchema; for example:
  ```
  schemagen sample.Address sample\package-info.java
  ```
- Use the @XmlType annotation namespace attribute to specify a namespace; for example:
  ```
  @XmlType(namespace="http://myNameSpace")
  ```

1. Locate the Java source files or Java class files to use in generating an XML schema file. Ensure that all classes referenced by your Java class files are contained in the classpath or are provided to the tool using the`-classpath/-cp` options.
2. Use the JAXB schema generator, schemagen command to generate an XML schema. The schema generator is located in the *app_server_root*\bin\ directory.

   ```
   app_server_root\bin\schemagen.bat myObj1.java myObj2.java
   ```

   ```
   app_server_root/bin/schemagen.sh myObj1.java myObj2.java
   ```

   ```
   app_server_root/bin/schemagen myObj1.java myObj2.java
   ```

   The parameters, `myObj1.java` and `myObj2.java`, are the names of the Java files containing the data objects. If `myObj1.java` or `myObj2.java` refer to Java classes that are not passed into the schemagen command, you must use the `-cp` option to provide the classpath location for these Java classes.Read about the schemagen command to learn more about this command and additional options that you can specify.
3. (Optional) Use JAXB program annotations defined in the `javax.xml.bind.annotations` package to customize the JAXB XML schema mappings.

4. (Optional) Configure the `location` property on the @XmlSchema annotation to indicate to the schema compiler to use an existing schema rather than generating a new one. For example,

```
@XmlSchema(namespace="foo")
package foo;
@XmlType
class Foo {
@XmlElement Bar zot;
}
@XmlSchema(namespace="bar",
location="http://example.org/test.xsd")
package bar;
@XmlType
class Bar {
...
}
<xs:schema targetNamespace="foo">
<xs:import namespace="bar"
schemaLocation="http://example.org/test.xsd"/>
<xs:complexType name="foo">
<xs:sequence>
<xs:element name="zot" type="bar:Bar" xmlns:bar="bar"/>
</xs:sequence>
</xs:complex
```

   the `location="http://example.org/test.xsd"` indicates the location on the existing schema to the schemagen tool and a new schema is not generated.

Results

Now that you have generated an XML schema file from Java classes, you are ready to marshal and unmarshal the Java objects as XML instance documents.

Note: The **schemagen** command does not differentiate the XML namespace between multiple @XMLType annotations that have the same @XMLType name defined within different Java packages. When this scenario occurs, the following error is produced:

```
Error: Two classes have the same XML type name ....
Use @XmlType.name and @XmlType.namespace to assign different names to them...
```

This error indicates you have class names or @XMLType.name values that have the same name, but exist within different Java packages. To prevent this error, add the @XML.Type.namespace class to the existing @XMLType annotation to differentiate between the XML types.

Example

The following example illustrates how JAXB tooling can generate an XML schema file from an existing Java class, Bookdata.java.

1. Copy the following `Bookdata.java` file to a temporary directory.

```
package generated;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import javax.xml.datatype.XMLGregorianCalendar;
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "bookdata", propOrder = {
    "author",
    "title",
    "genre",
    "price",
    "publishDate",
    "description"
})
public class Bookdata {

    @XmlElement(required = true)
    protected String author;
    @XmlElement(required = true)
    protected String title;
    @XmlElement(required = true)
    protected String genre;
    protected float price;
    @XmlElement(name = "publish_date", required = true)
    protected XMLGregorianCalendar publishDate;
    @XmlElement(required = true)
    protected String description;
    @XmlAttribute
    protected String id;

     public String getAuthor() {
        return author;
    }
    public void setAuthor(String value) {
        this.author = value;
    }
    public String getTitle() {
        return title;
    }

     public void setTitle(String value) {
        this.title = value;
    }


    public String getGenre() {
        return genre;
    }

    public void setGenre(String value) {
        this.genre = value;
    }


    public float getPrice() {
        return price;
    }


    public void setPrice(float value) {
        this.price = value;
    }


    public XMLGregorianCalendar getPublishDate() {
        return publishDate;
    }


    public void setPublishDate(XMLGregorianCalendar value) {
        this.publishDate = value;
```

```
    }


    public String getDescription() {
        return description;
    }


    public void setDescription(String value) {
        this.description = value;
    }


    public String getId() {
        return id;
    }


    public void setId(String value) {
        this.id = value;
    }

}
```

2. Open a command prompt.
3. Run the **schemagen** schema generator tool from the directory where you copied the `Bookdata.java` file.

```
app_server_root\bin\schemagen.bat Bookdata.java
```

```
app_server_root/bin/schemagen.sh Bookdata.java
```

```
app_server_root/bin/schemagen Bookdata.java
```

4. The XML schema file, `schema1.xsd` is generated:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="bookdata">
    <xs:sequence>
      <xs:element name="author" type="xs:string"/>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="genre" type="xs:string"/>
      <xs:element name="price" type="xs:float"/>
      <xs:element name="publish_date" type="xs:anySimpleType"/>
      <xs:element name="description" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Refer to the JAXB Reference implementation documentation for additional information about the **schemagen** command.

Using JAXB xjc tooling to generate JAXB classes from an XML schema file

Use Java Architecture for XML Binding (JAXB) xjc tooling to compile an XML schema file into fully annotated Java classes.

Before you begin

Develop or obtain an XML schema file.

About this task

Use JAXB APIs and tools to establish mappings between an XML schema and Java classes. XML schemas describe the data elements and relationships in an XML document. After a data mapping or binding exists, you can convert XML documents to and from Java objects. You can now access data stored in an XML document without the need to understand the data structure.

To develop Web services using a top-down development approach starting with an existing Web Services Description Language (WSDL) file, use the wsimport tool to generate the artifacts for your Java API for XML-Based Web Services (JAX-WS) applications or the Service Component Architecture (SCA) Java representations of your business service interfaces when starting with a WSDL file. After the Java artifacts for your application are generated, you can generate fully annotated Java classes from an XML schema file by using the JAXB schema compiler, xjc command-line tool. The resulting annotated Java classes contain all the necessary information that the JAXB runtime requires to parse the XML for marshaling and unmarshaling. You can use the resulting JAXB classes within Java API for XML Web Services (JAX-WS) applications or other Java applications such as SCA applications for processing XML data.

In addition to using the xjc tool from the command-line, you can invoke this JAXB tool from within the Ant build environments. Use the `com.sun.tools.xjc.XJCTask` Ant task from within the Ant build environment to invoke the xjc schema compiler tool.

1. Use the JAXB schema compiler, xjc command to generate JAXB-annotated Java classes. The schema compiler is located in the *app_server_root*\bin\ directory. The schema compiler produces a set of packages containing Java source files and JAXB property files depending on the binding options used for compilation.
2. (Optional) Use custom binding declarations to change the default JAXB mappings. Define binding declarations either in the XML schema file or in a separate bindings file. You can pass custom binding files by using the **-b** option with the xjc command.
3. Compile the generated JAXB objects. To compile generated artifacts, add the Thin Client for JAX-WS with WebSphere Application Server to the classpath.

Results

Now that you have generated JAXB objects, you can write Java applications using the generated JAXB objects and manipulate the XML content through the generated JAXB classes.

Example

The following example illustrates how JAXB tooling can generate Java classes when starting with an existing XML schema file.

1. Copy the following `bookSchema.xsd` schema file to a temporary directory.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="CatalogData">
     <xsd:complexType >
```

```
      <xsd:sequence>
        <xsd:element name="books" type="bookdata" minOccurs="0"
 maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="bookdata">
    <xsd:sequence>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="genre" type="xsd:string"/>
      <xsd:element name="price" type="xsd:float"/>
      <xsd:element name="publish_date" type="xsd:dateTime"/>
      <xsd:element name="description" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:schema>
```

2. Open a command prompt.
3. Run the JAXB schema compiler, xjc command from the directory where the schema file is located. The xjc schema compiler tool is located in the *app_server_root*\bin\ directory.

   ```
   app_server_root\bin\xjc.bat bookSchema.xsd
   ```

   ```
   app_server_root/bin/xjc.sh bookSchema.xsd
   ```

   Running the xjc command generates the following JAXB Java files:

   ```
   generated\Bookdata.java
   generated\CatalogdData.java
   generated\ObjectFactory.java
   ```

4. Use the generated JAXB objects within a Java application to manipulate XML content through the generated JAXB classes.

Refer to the JAXB 2.0 Reference implementation documentation for additional information about the xjc command.

Chapter 18. Defining and managing secure policy set bindings

You can specify abstract intents in the Service Component Architecture (SCA) composite file to achieve a quality of service for a service, reference, or secure connection by mapping intents to policy sets. You can also configure Web Service bindings to perform authentication using Lightweight Third-Party Authentication (LTPA) tokens.

Configuring Web service binding for SCA transport layer authentication

Use this task to specify abstract intents in the Service Component Architecture (SCA) composite file to achieve a quality of service for a service or reference. These intents must be mapped to policy sets that can satisfy the intents during deployment.

Before you begin

Before you begin this task, install Service Component Architecture (SCA) application.

About this task

Intents and policy sets can be used to configure Web service bindings to achieve quality of services (QoS).

1. Configure administrative and application security for the server.

   In order to secure the service so that it only accepts secure requests, and for the service to require authentication, administrative and application security must be enabled for the server. See Securing JAX-WS Web services using message-level security.

2. Configure the service to require transport authentication by specifying the authentication.transport intent on the <binding.ws> element.

   A component service can be configured to require transport authentication by specifying the ″authentication.transport″ intent on the <binding.ws> element.

   ```
   <service name="AccountService">
     <binding.ws
        requires="authentication.transport"
        ... />
   </service>
   ```

3. Configure the client to send a username and password by attaching the WSHTTPS default policy set to the <binding.ws> element.

   The wsPolicySet attribute can be used to specify policy sets at the composite, component, service, reference, and binding.ws levels in the SCA composite file. The actual attachment happens only at the binding.ws level and policy sets specified at other levels are inherited down to the binding level. For additional information on attaching policy sets to the <binding.ws> element and the inheritance rules, refer to mapping abstract intent to policy sets.

   After the policy set is attached to the client reference, it also requires assigning a client policy binding with the username and password provided in the HTTP transport binding to send with the request. Complete this task using the topic, configuring the HTTP transport policy, to configure the HTTP transport binding

to provide username and password. To assign policy set bindings, see the topic, defining and managing policy set bindings.

Results

When you finish this task, you have configured Web service binding to do SCA transport layer authentication.

What to do next

You can proceed to configuring other application specific bindings for your policy sets.

Configuring Web service binding to use SSL

Use this task to specify abstract intents in the Service Component Architecture (SCA) composite file to achieve a quality of service for secure connection using Secure Sockets Layer (SSL). The default SCA composite file is called *default.composite* and it is located in the META-INF level of the application structure. These intents must be mapped to policy sets that can satisfy the intents during deployment.

Before you begin

Before you begin this task, install a service application.

About this task

Intents and policy sets can be used to configure Web service bindings to achieve a secure connection.

1. Configure administrative and application security for the server.

   In order to secure the service so that it only accepts secure requests, and for the service to require authentication, administrative and application security must be enabled for the server. See Securing JAX-WS Web services using message-level security.

2. Configure the service to require a secure transport by attaching the WSHTTPS default policy set.

   Policy sets and bindings can be specified for SCA services and references using one of three different methods.

   - Specify a policy set and bindings directly in the composite file.
   - Attach a policy set during deployment using the addCompUnit command.
   - Attached or update a policy set during post deployment using the Web services policy set management panels in the administrative console.

   For additional information on each of the methods for attaching a policy set, see mapping abstract intents and managing policy sets. The code examples that are included in this task step and the next step use the composite file method to specify the WSHTTPS Default policy set.

   Attach the WSHTTPS default policy set and define the quality of service (QoS) namespace in the composite file.

```
<service name="AccountService">
  <binding.ws
     qos:wsPolicySet="WSHTTPS default"
     ... />
</service>
```

3. Configure the client to use SSL connection by attaching a policy set to the <binding.ws> element.

   The wsPolicySet attribute can be used to specify policy sets at the composite, component, service, reference, and binding.ws levels in the SCA composite file. The actual attachment happens only at the binding.ws level and policy sets specified at other levels are inherited down to the binding level. For additional information on attaching policy sets and the inheritance rules, refer to mapping abstract intent to policy sets. The following example illustrates the attachment of WSHTTPS default policy set to the <binding.ws> element.

   ```
   <reference name="AccountService">
     <binding.ws
        qos:wsPolicySet="WSHTTPS Default"
        ... />
   </reference>
   ```

   The WSHTTPS default policy set is a default policy set available in every server profile and it provides client-side SSL transport configuration. For additional information, see WSHTTPS default policy set.

   Important: The client must use an endpoint address of the form https://<host>:<secure-port> to contact the service.

Results

When you finish this task, you have configured Web service bindings to use SSL.

What to do next

You can proceed to configuring other application specific bindings for your policy sets.

Configuring Web service binding for LTPA authentication

Use this task to configure Web Service binding to perform authentication using Lightweight Third-Party Authentication (LTPA) tokens.

Before you begin

Before you begin this task, install Service Component Architecture (SCA) application.

About this task

Policy sets can be used to configure Web service bindings to perform authentication using LTPA tokens.

1. Configure the administrative and application security for the server.

   In order to secure the service so that it only accepts secure requests, and for the service to require authentication, administrative and application security must be enabled for the server. See Securing JAX-WS Web services using message-level security.

2. Configure the service to require message layer authentication by attaching the LTPA WSSecurity default policy set.

   To attach the LTPA WSSecurity default policy set, perform the task, mapping abstract intent to policy sets and policy management.

   In addition to attaching the policy set, you must configure the WS-Security policy to add a caller binding in order for the received subject to be propagated

to the thread. To update the default binding to support the caller function, open the administrative console and navigate to **Services > Policy sets > General provider policy set bindings** > **Provider sample > WS-Security > Callers**. Create a new Caller with the following values:

Name: Specify any name for this configuration
Caller identity local part: LTPAv2
Caller identity namespace URI: http://www.ibm.com/websphere/appserver/tokentype

For additional information on LTPA WSSecurity default policy set review the topic, WSSecurity default policy sets. Read also the article about configuring the WS-Security policy.

The following code is an example of configuring the service to support LTPA authentication.

```
<service name="AccountService">
  <binding.ws
     qos:wsPolicySet="LTPA WSSecurity default" qos:wsServicePolicySetBinding="Provider sample"
     ... />
</service>
```

3. Configure the client by attaching the LTPA WSSecurity default policy set to a reference.

   An example of how to attach the LTPA WSSecurity default policy set to a reference is shown in the code block in this task step. Attaching the LTPA WSSecurity default policy set to a reference by default propagates any existing LTPA tokens on the thread with the request. It is also possible to configure the policy set to create a token for a specific user and send that token with all requests. Refer to the article, WSSecurity default policy sets for detail information.

```
<reference name="AccountService">
  <binding.ws
     qos:wsPolicySet="LTPA WSSecurity default"
     ... />
</reference>
```

Results

When you finish this task, you have configured Web service bindings to do LTPA authentication.

What to do next

You can proceed to configuring other application specific bindings.

Chapter 19. Mapping abstract intents and managing policy sets

Use this task topic to specify abstract intents in the Service Component Architecture (SCA) composite file or with annotations to achieve a quality of service (QoS) for a service or reference when you are working with Web services policy sets and bindings. The intents in the composite file must be mapped to policy sets that can satisfy those intents during deployment to achieve the QoS that is required.

Before you begin

Before you begin this task, install an Service Component Architecture (SCA) application.

About this task

Use the composite file to specify intents and policy sets for Web services bindings to achieve quality of services (QoS).

1. Specify abstract intents at the <binding.ws> element of the service or reference level of the component.

   You can specify intents at any level in the composite file using a required attribute. The intent inheritance rules specified in the SCA Policy specification is enforced in the product. The following code example illustrates how to specify intents in a composite file.

   Specifying intents in a composite file

   ```
   <service name="AccountService" requires="authentication">
     <binding.ws
        requires="confidentiality.transport"
        ... />
   </service>
   ```

   Valid intents that can be mapped to a policy set are:

   confidentiality
   confidentiality.message
   confidentiality.transport
   integrity
   integrity.message
   integrity.transport
   authentication
   authentication.message
   authentication.transport
   propagatesTransaction

   The table provides a list of intents and the supported bindings to achieve QoS.

Table 40. Intents supported by different types of bindings

Intent	binding.ws	binding.ejb and binding.sca
authentication.message	Intent requires the attachment of a Web service policy set and policy binding that contains the WS–Security policy type	Not supported; CSIv2 can be configured to use basic auth or security token (LTPA, Kerberos)

Table 40. Intents supported by different types of bindings (continued)

Intent	binding.ws	binding.ejb and binding.sca
confidentiality.message and integrity.message	Both intents require the attachment of a Web service policy set and policy binding that contains the WS–Security policy type	Not supported
authentication.transport	Basic auth only. Reference requires the attachment of a Web service policy set that contains the HTTPTransport policy type. Service does not require any attachments.	Intent is not supported. CSIv2 can be configured to use client certificates for authentication.
confidentiality.transport and integrity.transport	Requires the attachment of a Web service policy set that contains the SSLTransport policy type	Intent is not supported. CSIv2 can be configured to require SSL.
propagatesTransaction	Requires the attachment of a Web service policy set that contains the WS–Transaction policy type	Supported; no configuration required

Mapping of these intents to a policy set can be done during deployment using the interactive addCompUnit command or the administrative console. The Policy Set column in the administrative console page provides hints as to which policy sets might best satisfy the intents. You can choose to attach one of the suggested default policy sets or attach any other policy set that is defined in the product. Attaching no policy set is a valid choice. This might be the case when there are no satisfying policy sets or the intents are not valid.

2. Attach policy sets and bindings.

   Important:

   Annotations are not supported with Web services policy sets.

   The policy sets referenced in this article are defined for Web services and apply to Web service bindings. For an overview of policy sets, read Web services policy sets.

   You can attach policy sets and bindings at any of the levels in a composite file. The product provides some default policy sets, as well as a sample service and reference binding for the client. You can create additional policy sets and bindings using the Web services administrative console or scripting functionality. Refer to creating policy sets using the administrative console or creating policy set attachments using the wsadmin tool.

   Policy sets and bindings can be specified for SCA services and references using one of three different methods.

   - Specify a policy set and bindings directly in the composite file.
   - Attach a policy set during deployment using the addCompUnit command.
   - Attached or update a policy set during post deployment using the Web services policy set management panels in the administrative console.

   Policy set management using commands

   The addCompUnit command can be run in interactive mode to attach policy sets. After navigating to the AttachPolicySet step, the policy set to be attached can be specified for different resource ID's.

   When running the addCompUnit command in batch mode, the policy set can be directly attached to a resource ID by using the -AttachPolicySet step. The syntax of the command is:

```
AdminTask.addCompUnit('[-blaID ... -AttachPolicySet[[<resourceID> .* .* .*  <Policy set to attach>]]]')
```

   The following are the valid resource ID's:

```
<composite name>
<composite name>/<component name>
<composite name>/<component name>/<service name>
<composite name>/<component name>/<service name>/binding.ws
<composite name>/<component name>/<reference name>
<composite name>/<component name>/<reference name>/binding.ws
```

The following is an example of the command:

```
AdminTask.addCompUnit ('[-blaID myBLA -cuSourceID echoService.jar -AttachPolicySet
 [[EchoServiceWSComposite/EchoServiceWSComponent/EchoService/binding.ws .* .* .* "WSHTTPS default"]]]')
```

The command above attaches the WSHTTPS default policy set to the binding.ws under EchoService. The policy is fine tuned using the provider sample binding. The policy set attached during deployment can be changed at post-deployment using the administrative console or commands.

Policy set management using the administrative console

You can also attach policy sets to your business level applications using the administrative console during deployment and post deployment. During deployment, the policy set can be configured in the Attach policy set panel that shows up when you are adding a composition unit. To view the Attach policy set panel, you must be deploying a business-level application that uses Web services. This panel does not exist in post deployment. During post deployment, the policy can be configured using the service provider and the service client panels under the Services menu. For additional information on using the administrative console to attach policy sets, see attach policy set settings document.

Policy set management using composite file

You can specify wsPolicySet attribute for policy sets at the composite, component, service, reference, and binding.ws levels. The actual attachment happens only at the binding.ws level and policy sets specified at the other levels are inherited down to the bindings level. The inheritance rules are described in the next task step. Similar to the wsPolicySet, you can specify the wsServicePolicySetBinding and wsReferencePolicySetBinding attributes. First create the policy sets and the bindings using Web services administrative tools, and then attach the policy set using one of the three methods described in this task step.

The following is an example of a composite file with a wsPolicySet mapping.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           ...
           xmlns:qos="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
           name="EchoServiceWSComposite">

    <component name="EchoServiceWSComponent">
        ...
        <service name="EchoService">
            <binding.ws
                  qos:wsPolicySet="WSHTTPS default" qos:wsServicePolicySetBinding="Provider sample"

                  ... />
        </service>

    </component>
</composite>
```

3. Define policy set inheritance.

   Policy sets are inherited in a top down format. However, a policy set specified at a lower level has precedence over what it inherits. For example, in the sample composite file shown in this task step, the policy set attached for the

binding for EchoService1 is LTPA WSSecurity default but the bindings at EchoService2 inherits the WSHTTPS default policy set from the composite level.

Defining policy set inheritance

```
<?xml version="1.0" encoding="UTF-8"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
           ...
                    xmlns:qos="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
          name="EchoServiceWSComposite" qos:wsPolicySet="WSHTTPS default">
    <component name="EchoServiceWSComponent">
        ...
        <service name="EchoService1">
            <binding.ws
                  qos:wsPolicySet="LTPA WSSecurity default"
                  ... />
        </service>
        <service name="EchoService2">
            <binding.ws
                   ... />
        </service>
    </component>
</composite>
```

Results

When you finish this task, you have mapped abstract intents to policy sets and attached policy sets to your SCA service artifact.

What to do next

You can proceed to configure application specific bindings.

Attached deployed assets collection

Use this page to view assets that are attached to a policy set, detach or replace a policy set.

To view this administrative console page, complete the following actions:

1. Click **Services** → **Policy sets** → **Application policy sets** → ***policy_set_name***.
2. Click **Attached deployed assets** link in the Additional Properties section.

Name

Specifies a list of deployed assets. The deployed asset names that are displayed in the Name column are either attached explicitly to the specified policy set or have service resources attached to this policy set.

To alter the policy set that is attached to a deployed asset, select a deployed asset, and click a button to enable the following actions:

Table 41. Button descriptions

Button	Resulting action
Detach Policy Set	Detaches the current policy set from the selected deployed asset or deployed assets. This action also detaches any attached deployed asset service resources. This action does not detach other policy sets from the deployed asset or deployed asset service resources. To perform this action, select a resource in the **Name** column, and click **Detach Policy Set**. This action detaches the policy set from the selected asset.

Table 41. Button descriptions (continued)

Button	Resulting action
Replace Policy Set	Displays a list of policy sets that can be attached to the selected deployed asset or deployed assets and any contained attached service resources. The policy set is replaced with the one selected. This action does not replace other policy sets that are attached to resources in the deployed asset that you select. To perform this action, select a resource in the **Name** column, and click **Replace Policy Set**. This action enables you to select a different policy set to attach to the selected deployed asset in place of the current policy set.

Type

Specifies the type of deployed asset, such as a composition unit (CU) or a Java Platform, Enterprise Edition (Java EE) application.

Chapter 20. Administering asynchronous beans

Configuring work managers

A work manager acts as a thread pool for application components that use asynchronous beans. Use the administrative console to configure work managers.

Before you begin

If you are not familiar with work managers, refer to the Work managers conceptual topic.

About this task

The work manager service is always enabled. In previous versions of the product, the work manager service could be disabled using the administration console or configuration service. The work manager service configuration objects are still present in the configuration service, but the enabled attribute is ignored.

You can define multiple work managers for each cell. Each work manager is bound to a unique place in Java Naming and Directory Interface (JNDI).

Important: The work manager service is only supported from within the Enterprise Java Beans (EJB) Container or Web Container. Looking up and using a configured work manager from a Java Platform, Enterprise Edition (Java EE) application client container is not supported.

1. Start the administrative console.
2. Select **Resources > Asynchronous beans > Work managers**.
3. Specify a **Scope** value and click **New**.
4. Specify the required properties for work manager settings.
   Scope The scope of the configured resource. This value indicates the location for the configuration file.
   Name The display name for the work manager.
   JNDI Name
   The Java Naming and Directory Interface (JNDI) name for the work manager. This name is used by asynchronous beans that need to look up the work manager. Each work manager must have a unique JNDI name within the cell.
   Number of Alarm Threads
   The maximum number of threads to use for processing alarms. A single thread is used to monitor pending alarms and dispatch them. An additional pool of threads is used for dispatching the threads. All alarm managers on the asynchronous beans associated with this work manager share this set of threads. A single alarm thread pool exists for each work manager, and all of the asynchronous beans associated with the work manager share this pool of threads.
   Minimum Number Of Threads
   The number of threads to be kept in the thread pool, created as needed.
   Maximum Number Of Threads
   The maximum number of threads to be created in the thread pool. The maximum number of threads can be exceeded temporarily if the

Growable check box is selected. These additional threads are discarded when the work on the thread completes.

Thread Priority
The priority to assign to all threads in the thread pool.

Every thread has a priority. Threads with higher priority are run before threads with lower priority. For more information about how thread priorities are used, see the javadoc for the setPriority method of the java.lang.Thread class in the Java Standard Edition specification.

5. [Optional] Specify a **Description** and a **Category** for the work manager.
6. [Optional] Select the **Service Names** (Java EE contexts) on which you want this work manager to be made available. Any asynchronous beans that use this work manager then inherit the selected Java EE contexts from the component that creates the bean. The list of selected services also is known as the ″sticky″ context policy for the work manager. Selecting more services than are actually required might impede performance.

   Other optional fields include:

   Work timeout
   Specifies the number of milliseconds to wait before a scheduled work object is released. If a value is not specified, then the timeout is disabled.

   Work request queue size
   Specifies the size of the work request queue. The work request queue is a buffer that holds scheduled work objects and can be a value of 1 or greater. The thread pool pulls work from this queue. If you do not specify a value or the value is 0, the queue size is managed automatically. When the queue size is managed automatically, it is computed as the (*minimum_number_of_threads* + *maximum_number_of_threads*) / 2. If this value computes to a zero value, a queue size of 1 is used. Large values can consume significant system resources.

   Work request queue full action
   Specifies the action taken when the thread pool is exhausted, and the work request queue is full. This action starts when you submit non-daemon work to the work manager. If set to FAIL, the work manager API methods creates an exception instead of blocking.

   Default transaction class
   Specifies the transaction class name used to classify work run by this work manager instance when the z/OS Work Load Manager Service class information is not contained in the work context information.

   Daemon transaction class
   Specifies the transaction class name used to classify ″daemon″ work initiated by this work manager instance.
7. Save your configuration.

Results

The work manager is now configured and ready for access by application components that need to manage the start of asynchronous code.

Configuring Work managers for one-way operations

You can configure Work managers for one-way operations. In Feature Pack for SCA, Work manager configuration is supported only at the component service level. This capability is supported for default and Web services binding.

Before you begin

Asynchronous service dispatches run on a different thread than the service invoker. The thread pool that services these dispatches must be configurable so that the service thread pool can be adjusted, based on workload variations or some other policy. The application server has configurable thread pools, called Work managers, that can be used as the means for providing an asynchronous thread pool to an SCA service.

This topic applies to non-blocking (one-way) operations only. The Java composite definition (`.composite` file) for the component service that has the one-way operation, enabling the Work manager to be configured independently.

Specifying one Work manager per component service is optional. If you do not specify a Work manager component service that has the one-way operation, the runtime environment uses a default Work manager that is configured at the server, node, or cell level.

Important:

- Work manager configuration is supported only at the component service level.
- This feature is supported for default and Web services bindings.

1. Configure Asynchronous beans Work manager. In the administrative console, click **Resources** → **Asynchronous beans** → **Work managers**. Create a new Work manager. Remember the JNDI name provided.
2. Restart the server.
3. Assemble a composite with one-way operation.
4. Define the composite definition by adding the WorkManager namespace definition, `http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06`, and specify the JNDI name of the Work manager configured previously under component service, *<workManager ../>*, element.
5. Use the administrative console to define a Work manager and assign it a JNDI name.
6. Restart the server.
7. Develop a service component with a non-blocking, one-way operation:
8. Specify Work manager for component service.
   a. Add the WorkManager namespace definition, `http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06`, to the composite definition (`.composite` file), for example:

      ```
      <composite xmlns="http://www.osoa.org/xmlns/sca/1.0" xmlns:dbsdo=
      "http://tuscany.apache.org/xmlns/sca/databinding/sdo/1.0"
      xmlns:wm="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06"
      name="ExampleComposite">
      ```

   b. Add the JNDI name of the Work manager to the component service, *<work manager..../>*, element, for example:

```
<component name="Example_Component">
       <service name="Example_Service">
            <wm:workManager value="sca/example"/>
        </service>
        <implementation.java class="test.sca.binding.sca.ExampleImpl"/>
 </component>
```

9. Deploy the service component.
10. Change the deployed service Work manager setting to a different JNDI name.
11. Restart the business application.
12. Change the configuration of the Work manager that you created.
13. Restart the business application.

Results

Service operation dispatches occur in the thread pool of the configured Work manager.

Example

Sample composite definition with Work manager configuration: In this sample, the first component runs with the configured Work manager, which has JNDI name `sca/test2`. The second component does not have a Work manager setting, even though it is one-way, hence the second component runs with the default Work manager, which is configured per server, node, or cell level. The third component runs with its Work manager with JNDI name `sca/test4`.

```
<?xml version="1.0" encoding="UTF-8"?>
http://www.osoa.org/xmlns/sca/1.0
targetNamespace="http://samples.myco.com/oneway"
 xmlns:wm="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06" name="Composite_2">
    <component name="Component_2">
       <service name="Component_2">
            <wm:workManager value="sca/test2"/>
        </service>
        <implementation.java class="test.sca.binding.sca.oneway.Component_2Impl"/>
        <reference name="component_3"  target="Component_3"/>
    </component>
    <component name="Component_3">
        <implementation.java class="test.sca.binding.sca.oneway.Component_3Impl"/>
        <reference name="component_4"  target="Component_4"/>
    </component>
    <component name="Component_4">
            <service name="Component_4">
                <wm:workManager value="sca/test4"/>
            </service>
            <implementation.java class="test.sca.binding.sca.oneway.Component_4Impl"/>
    </component>
</composite>
```

Sample service interface with one-way operation:

```
import org.osoa.sca.annotations.OneWay;
import org.osoa.sca.annotations.Remotable;

@Remotable
public interface Component_2 {
 @OneWay
 public void test(Message inputText);
 }
```

Configuring the default SCA Work manager for the SCA layer

This topic describes how to configure the default SCA Work manager for the entire SCA layer. Create this Work manager if you need to tune or monitor the thread pool for asynchronous invocation at the entire SCA layer level. If this Work manager is not configured, SCA layer creates one automatically.

About this task

Create the SCA Work manager, as follows:

1. Create a new Work manager. In the administrative console, click **Resources > Asynchronous beans > Work managers**. Remember the JNDI name provided.
2. Configure SCA to use the newly-created work manager. In the administrative console, click **Application servers > *server* > Process definition > Java virtual machine > Custom properties**.
3. Add a new property named `SCAWorkManager` with the value as the JNDI name that was provided in step one.
4. Save and restart the server.

Results

Service operation dispatches occur in the thread pool of the configured Work manager.

Chapter 21. Transaction support in WebSphere Application Server

Support for transactions is provided by the transaction service within WebSphere Application Server. The way that applications use transactions depends on the type of application component.

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, during the processing of an SQL COMMIT statement, the database manager atomically commits multiple SQL statements to a relational database. 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). If a transaction involves multiple resource managers, for example multiple database managers, an external transaction manager is required to coordinate the individual resource managers. A transaction that spans multiple resource managers is referred to as a *global transaction*. WebSphere Application Server is a transaction manager that can coordinate global transactions, can be a participant in a received global transaction, and can also provide an environment in which resource manager local transactions can run.

The way that applications use transactions depends on the type of application component, as follows:

- A session bean can use either container-managed transactions (where the bean delegates management of transactions to the container) or bean-managed transactions (component-managed transactions where the bean manages transactions itself).
- Entity beans use container-managed transactions.
- Web components (servlets) and application client components use component-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 transaction managers that support the CORBA Object Transaction Service (OTS) protocol or Web Service Atomic Transaction (WS-AtomicTransaction) protocol. WebSphere Application Server also participates in transactions imported through Java EE Connector 1.5 resource adapters. You can also configure WebSphere applications to interact with databases, JMS queues, and JCA connectors through their *local transaction* support, when you do not require distributed transaction coordination.

In addition to supporting the coordination of XAResource-based resource managers, WebSphere Application Server for z/OS supports the coordination of resource managers through RRS (z/OS resource recovery services). RRS-compliant resource managers include DB2®, WebSphere MQ, IMS™, and CICS®. IBM WebSphere Application Server for z/OS can coordinate 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.

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 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.

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 complete any updates, the one-phase commit resource does not have to be prepared.
- In scenarios where there is only a single one-phase commit resource provider that participates in the transaction with one or more two-phase commit resource providers and where *last participant support* is enabled. Last participant support 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. For more information about last participant support, see Using one-phase and two-phase commit resources in the same transaction.

The ActivitySession service provides an alternative unit-of-work (UOW) scope to that provided by global transaction contexts. It is a distributed context that can be used to coordinate multiple one-phase resource managers. The WebSphere EJB container and deployment tooling support ActivitySessions as an extension to the Java EE programming model. Enterprise beans can be deployed with lifecycles that are influenced by ActivitySession context, as an alternative to transaction context. An application can then interact with a resource manager for the period of a client-scoped ActivitySession, rather than only the duration of an EJB method, and have the resource manager local transaction outcome directed by the ActivitySession. For more information about ActivitySessions, see Using the ActivitySession service.

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.

SCA transaction intents

Service Component Architecture (SCA) provides declarative mechanisms in the form of *intents* for describing the transactional environment required by components.

This topic covers:

- "Using a global transaction"
- "Using local transaction containment" on page 176
- "Transaction intent default behavior" on page 177
- "Managed local or global transactions with JDBC data sources and IBM i" on page 178

Using a global transaction

Components that use a synchronous interaction style can be part of a single, distributed ACID transaction within which all transaction resources are coordinated to either atomically commit or roll back. This is specified by using the **managedTransaction.global** intent in the `requires` attribute of the `<implementation.java>` element as shown below.

```
<component name="DataAccessComponent">
    <implementation.java class="example.DataAccessImpl"
         requires="managedTransaction.global"/>
</component>
```

It is possible to control whether a component's service runs under its client's global transaction by specifying either the **propagatesTransaction** or **suspendsTransaction** intent on the component's `<service>` element.

- **propagatesTransaction** - The service runs under its client's global transaction. If the client is not running in a global transaction or chose not to propagate its global transaction, the service runs in its own global transaction.
- **suspendsTransaction** - The service runs in its own global transaction separate from the client transaction.

It is also possible to control whether a component global transaction is propagated to a referenced service by specifying either the **propagatesTransaction** or **suspendsTransaction** intent on the component *<reference>* element.

- **propagatesTransaction** - The component's global transaction is made available to the referenced service. The referenced service might or may not use this transaction depending on how it is configured.
- **suspendsTransaction** - The component's global transaction is not made available to the referenced service.

Transaction context is never propagated on `@OneWay` methods. The SCA run time ignores **propagatesTransaction** for OneWay methods.

The following example shows the use of the **managedTransaction.global**, **propagatesTransaction**, and **suspendsTransaction** intents. The **DataUpdateComponent** runs in its own global transaction, not in its client's transaction, because **suspendsTransaction** is specified on its `<service>` element. Its

global transaction is propagated to the referenced service **DataAccessComponent** because **propagatesTransaction** is specified on its `<reference>` element.

```
<component name="DataUpdateComponent">
    <implementation.java class="example.DataUpdateImpl"
            requires="managedTransaction.global"/>
    <service name="DataUpdateService"
            requires="suspendsTransaction"/>
    <reference name="myDataAccess" target="DataAccessComponent"
            requires="propagatesTransaction"/>
</component>
```

Propagating transactions over the Web service binding requires the use of a WebSphere policy set that contains the WS-Transaction policy type. You can set up this policy set in one of the following ways:

- You can import the WSTransaction policy set that is provided with the product.
- You can create your own policy set and include the WS-Transaction policy type.

The following example assumes the use of the WSTransaction policy set.

```
<composite name="WSDataUpdateComposite"
  xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:ws="http://www.ibm.com/xmlns/prod/websphere/sca/1.0/2007/06">
     <component name="WSDataUpdateComponent">
        <implementation.java class="example.DataUpdateImpl"
             requires="managedTransaction.global"/>
        <service name="DataUpdateService"
             requires="propagatesTransaction">
             <binding.ws ws:wsPolicySet="WSTransaction"/>
        </service>
        <reference name="myDataBuddy" target="DataBuddyComponent"
             requires="propagatesTransaction">
             <binding.ws ws:wsPolicySet="WSTransaction"/>
        </reference>
    </component>
</composite>
```

Tip: Transaction propagating might not result in a managed connection. Use a qualifying Java EE module for a managed connection and connection sharing.

Using local transaction containment

Business logic might have to access transactional resource managers without the presence of a global transaction. A component can be configured to run under local transaction containment (LTC). The SCA runtime starts an LTC before dispatching a method on the component and completes the LTC at the end of the method dispatch. The component's interactions with resource providers (such as databases) are managed within resource manager local transactions (RMLTs). A resource manager local transaction (RMLT) represents a unit of recovery on a single connection that is managed by the resource manager.

The local transaction containment policy is configured by using an intent. There are two choices:

- **managedTransaction.local** - Use this intent when each interaction with a resource manager should be part of an extended local transaction that is committed at the end of the method. The SCA runtime wraps interactions with each resource manager in a resource manager local transaction (RMLT). The SCA runtime commits each RMLT at the end of method dispatch, unless an unchecked exception occurs, in which case the SCA runtime aborts each RMLT. The component might not use resource manager commit/rollback interfaces or set AutoCommit to `true`. If multiple resource managers are used, the RMLTs are committed independently so it is possible for some to fail and some to succeed. If this behavior is not what you want, use a global transaction.

- **noManagedTransaction** - The SCA runtime does not wrap interactions with resource managers in a RMLT. The component implementation manages the start and end of its own RMLTs or gets AutoCommit behavior (which commits after each use of a resource) by default. The component must complete any RMLTs before the end of the method dispatch otherwise the SCA runtime will abort them.

The intent is specified by using the requires attribute on the <*implementation.java*> element. An example is shown below.

```
<component name="DataAccessLocalComponent">
    <implementation.java class="example.DataAccessImpl"
          requires="managedTransaction.local"/>
</component>
```

A local transaction cannot be propagated from one component to another. It is an error to specify **propagatesTransaction** on a component's <*service*> if the component uses the **managedTransaction.local** or **noManagedTransaction** intent.

Rollback

The SCA run time performs a rollback under the following circumstances:

- When **managedTransaction.global** is used, the SCA run time performs a rollback if the component method that started the global transaction throws an unchecked exception. An unchecked exception is a subclass of java.lang.RuntimeException or java.lang.Error. A checked exception does not force a rollback.
- When **managedTransaction.local** is used, the SCA run time performs a rollback if the component method throws an unchecked exception. An unchecked exception is a subclass of java.lang.RuntimeException or java.lang.Error. A checked exception does not force a rollback.
- When **noManagedTransaction** is used, the SCA run time performs a rollback of any RMLT that has not been committed by the component method, regardless of whether the method throws an exception or not.

When **managedTransaction.global** or **managedTransaction.local** is used, the business logic can force a rollback by using the UOWSynchronization interface.

```
com.ibm.websphere.uow.UOWSynchronizationRegistry uowSyncRegistry =
    com.ibm.wsspi.uow.UOWManagerFactory.getUOWManager();
        uowSyncRegistry.setRollbackOnly();
```

Transaction intent default behavior

If transactional intents are not specified, the default behavior is vendor-specific. If a transactional intent in not specified for the implementation, the default is **managedTransaction.global**. If a transactional intent is not specified for a service or reference, the default is **suspendsTransaction**. It is recommended to specify the required intents rather than to rely on default behavior so that the application is portable.

Using @Requires annotation to specify transaction intents

You can also specify transaction intents in the implementation class by using the `@Requires` annotation. The general form of the annotation is:

```
@Requires("{http://www.osoa.org/xmlns/sca/1.0}intent")
```

For example, you can use the following in the implementation class:

```
@Requires("{http://www.osoa.org/xmlns/sca/1.0}managedTransaction.global")
```

You can specify required intents on various elements, including the composite, component, implementation, service and reference elements. An element inherits the required intents of its parent element except when they conflict. For example, if a composite element requires `managedTranaction.global` and a component element requires `managedTransaction.local`, then the component uses `managedTransaction.local`.

Managed local or global transactions with JDBC data sources and IBM i

If you use managed local transaction intent or global transaction intent in SCA composites, this will turn off autocommit when you use JDBC on the IBM i platform. If multiple SQL jobs need to lock the same table for an update, the following exception will appear:

```
SQLException: com.ibm.db2.jdbc.app.DB2DBException: Row or object WAREHOUSE in CBIVP type *FILE in use
```

The first SQL job that locks the table never commits the transaction, and the lock is never released. Global and local transactions for SCA are not supported when you use JDBC to connect to resources, so you must use noManagedTransaction intents. Change the following in the SCA composite files to switch to the noManagedTransaction intent:

1. Change

   ```
   requires="managedTransaction.local"
   ```

   and

   ```
   requires="managedTransaction.global"
   ```

   to requires=″noManagedTransaction″
2. Change

   ```
   requires="propagatesTransaction"
   ```

   to

   ```
   requires="suspendsTransaction"
   ```

Chapter 22. Dynamic cache service eviction policies

Eviction policies using the disk cache garbage collector

The disk cache garbage collector is responsible for evicting objects out of the disk cache, based on a specified eviction policy.

The garbage collector keeps a certain amount of space on disk available, which is governed by the configuration attribute that limits the amount of disk space that is used for caching objects. To enable the eviction policy, enable the Limit disk cache size in GB and/or Limit disk cache size in entries options in the administrative console.

The garbage collector is triggered when the disk space reaches a specified high threshold (a percentage of the Limit disk cache size in entries or in GB) and evicts objects, based on the eviction policy, from the disk in the background until the disk cache size reaches a specified low threshold (a percentage of the Limit disk cache size in entries or in GB). Eviction triggers when one or both of the high thresholds is reached for Limit disk cache size in GB and Limit disk cache size in entries. The supported policies are:

- None: This is the default policy. Objects are evicted only when they expire, or if they are invalidated.
- Random: The expired objects are removed first. If the disk size still has not reached the low threshold limit, objects are picked from the disk cache in random order and removed until the disk size reaches a low threshold limit.
- Size: The expired objects are removed first. If the disk size still has not reached the low threshold limit, then largest-sized objects are removed until the disk size reaches a low threshold limit.

Limit disk cache size in GB and High Threshold determines when to trigger eviction and when the disk cache is considered near full. It is computed as a function of the user-specified limit. If the specified limit is *10* GB (3 GB is the minimum), the cache subsystem initially creates three files that can grow to 1 GB in size for cache data, dependency ID information, and template information. Each time more space is needed to contain cache data, dependency ID information, or template information, a new file is created. Each of these files grow in 1 GB increments until the total number of files that are created is equal to disk cache in size in GB (in this case ten). Although the initial size of the new file may be much smaller than 1 GB, the dynamic cache service always rounds up to the next GB.

Eviction triggers when the cache data size reaches the high threshold and continues until the cache data size reaches the low threshold. Calculation of cache data size is dynamic. The following formula describes how to calculate the actual cache data size limit:

```
cache data size limit = disk cache size (in GB) - number of dependency files per GB - number of template files
```

When the cache data size limit is defined, the trigger point is calculated as follows:

```
eviction trigger point = cache data size limit * high threshold
size of evicted entries = cache data size * (high threshold - low threshold)
```

Consider the following scenarios:

- **Scenario 1**

– Disk cache size in GB = 10 GB
– High threshold = 90%
– Low Threshold = 80%

Initially, there is one file for dependency ID and template ID.

```
cache data size limit = 10-(1+1) = 8 GB
eviction trigger point = 8 * 90% = 7.2 GB
size of evicted entries = 8 * (90% - 80%) = 0.8 GB
```

In the above scenario, eviction starts when the data cache size reaches 7.2 GB and continues until the cache size is 6.4 GB (7.2 - 0.8).

- **Scenario 2**

  In scenario 1, if the dependency files grow to more than 1 GB, an additional dependency file generates. The eviction trigger point launches dynamically as follows:

```
cache data size limit = 10 - (2+1) = 7GB
eviction trigger point = 7 * 90% = 6.3GB
size of evicted entries = 7 * (90% - 80%) = 0.7GB
```

  In the above scenario, eviction starts when the data cache size reaches 6.3 GB, and continues until the cache size in 5.6 GB (6.3 - 0.7).

Disk cache eviction for limit disk cache size in entries. Consider the following scenario:

- Disk cache size in entries = 100000
- High threshold = 90%
- Low threshold = 80%

```
eviction trigger point = 100000 * 90% = 90000
number of entries evicted = 100000 * (90% - 80%) = 10000
```

In this scenario, eviction starts when the number of cache entries reaches 90000 and 10000 entries are evicted from the cache.

Example: Caching Web services

This topic includes examples of building a set of cache policies and SOAP messages for a Web services application.

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 a good candidate for caching. In this example the SOAP message is cached and a timeout is placed on its entries to guarantee the quotes it returns are current.

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>
```

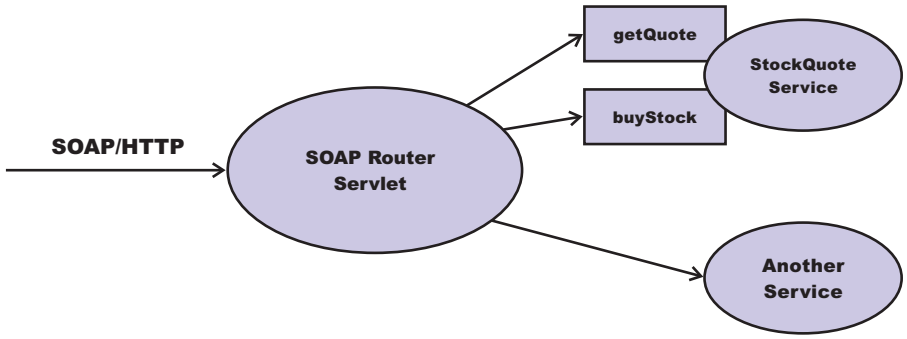
The SOAPAction HTTP header in the request is defined in the SOAP specification and is used by HTTP proxy servers to dispatch requests to particular HTTP servers. WebSphere Application Server dynamic cache can use this header in its cache policies to build IDs without having to parse the SOAP message.

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:buyStock>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The following 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 the 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>
  </component>
 </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>
  </component>
 </cache-id>
</cache-entry>
</cache>
```

Chapter 23. Using PassByReference optimization in SCA applications

Support exists for the @AllowsPassByReference annotation, which can be used to bypass marshaling and unmarshaling when a client invokes a service located in the same JVM over a remote interface.

About this task

Typically, a performance intensive aspect of service invocations is data marshaling and unmarshaling. Though invocation over a local interface always results in pass-by-reference semantics so that no data is copied, an invocation over a remotable interface entails pass-by-value semantics, which typically results in copying of the data which can be expensive.

Note: The SCA default binding provides the @AllowsPassByReference as an optimization that you can use on your service implementation at the class level or at the individual method level.

In placing the @AllowsPassByReference annotation on the service implementation class or methods, the implementor agrees not to modify the data in a way that would violate the pass-by-value semantics. This allows both client and service to assume they are working with their own copy of the data even though the runtime environment has optimized to not perform the actual data serialization and deserialization, to save this expense.

Parameters, return types, and business exceptions are passed by reference if the service implementation class has the @AllowsPassByReference annotation defined at the class level or individual method level.

More specifically, the PassByReference optimization is performed when all of the following are true:

- Client and service have been targeted to the same JVM.
- The invocation is over the default binding.
- Both client and service are part of the same business-level application
- @AllowsPassByReference is present. Either the service implementation is a Java implementation with an appropriate @AllowsPassByReference annotation, or a composite implementation, ultimately recursively implemented in terms of such an @AllowsPassByReference-annotated Java implementation.
- All input, output, and exception types have the same package-qualified class names and can be loaded by a classloader common to or shared by both client and service.

1. To enable PassByReference optimization for SCA applications, ensure all classes that you want to optimize are loaded by the same classloader. This is enabled in the Feature Pack for SCA by using the SCA contribution import and export support.
2. Create a Java archive (JAR) file that contains all classes that are loaded by the same classloader during both client and service execution.
3. Add an *sca-contribution.xml* file to the META-INF directory in the JAR.

See the OSOA Assembly specification for information on *sca-contribution.xml*. The contribution definition must contain an *export.java* statement that exports all packages contained in the JAR that are accessed by either the client or service JAR file. For example:

```
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://test.sca.scafp/pbr/shared/java">
    <export.java package="com.ibm.sample.interface"/>
    <export.java package="com.ibm.sample.types"/>
</contribution>
```

If the client and service JAR files are not already using an *sca-contribution.xml* file, update these files to use a contribution definition that imports the packages that are exported by the shared library. For example, the contribution files for the client and service that access the previous shared contribution might look like this:

```
Client:
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://test.sca.scafp/pbr/shared"
    xmlns:pbr="http://test.sca.scafp/pbr/shared">
    <deployable composite="pbr:PassByRef.SharedClient"/>
    <import.java package="com.ibm.sample.interface"/>
    <import.java package="com.ibm.sample.types"/>
</contribution>

Service:
<contribution xmlns="http://www.osoa.org/xmlns/sca/1.0"
    targetNamespace="http://test.sca.scafp/pbr/shared"
    xmlns:pbrsh="http://test.sca.scafp/pbr/shared">
    <deployable composite="pbrsh:PassByRef.SharedService"/>
    <import.java package="com.ibm.sample.interface"/>
    <import.java package="com.ibm.sample.types"/>
</contribution>
```

4. Deploy the SCA application

   Add the client, service, and shared contributions as an asset into the WebSphere repository. This can occur in any order, however you must add the shared contribution as an asset before you can add either the client or service asset as a composition unit to a business-level application. Only the client and service assets need to be added as composition units to your business-level applications. During the add composition unit operation for both the client and the service, the shared asset is automatically added to the business-level application as a shared library.

   Clients that are deployed outside of a business-level application cannot use the PassByReference optimization to invoke SCA services deployed inside a business-level application. For example, a user-created Web archive (WAR) file using the default binding cannot be installed into a business-level application, and therefore a WAR-hosted client might not participate in the PassByReference optimization. The PassByReference optimization is supported only between JAR files. You must install both service JAR files on the same business-level application.

Results

You have enabled PassByReference optimization for SCA applications.

Chapter 24. Directory conventions

References in product information to *app_server_root*, *profile_root*, and other directories infer specific default directory locations. This topic describes the conventions in use for WebSphere Application Server.

Default product locations - z/OS

app_server_root
Refers to the top directory for a WebSphere Application Server node.

The node may be of any type—application server, deployment manager, or unmanaged for example. Each node has its own *app_server_root*. Corresponding product variables are was.install.root and WAS_HOME.

The default varies based on node type. Common defaults are *configuration_root*/AppServer and *configuration_root*/DeploymentManager.

configuration_root
Refers to the mount point for the configuration file system (formerly, the configuration HFS) in WebSphere Application Server for z/OS.

The *configuration_root* contains the various *app_server_root* directories and certain symbolic links associated with them. Each different node type under the *configuration_root* requires its own cataloged procedures under z/OS.

The default is /wasv7config/*cell_name*/*node_name*.

plug-ins_root
Refers to the installation root directory for Web Server plug-ins.

profile_root
Refers to the home directory for a particular instantiated WebSphere Application Server profile.

Corresponding product variables are server.root and user.install.root.

In general, this is the same as *app_server_root*/profiles/*profile_name*. On z/OS, this will be always be *app_server_root*/profiles/default because only the profile name "default" is used in WebSphere Application Server for z/OS.

smpe_root
Refers to the root directory for product code installed with SMP/E.

The corresponding product variable is smpe.install.root.

The default is /usr/lpp/zWebSphere/V7R0.

Default product locations - IBM i

These file paths are default locations. You can install the product and other components in any directory where you have write access. You can create profiles in any valid directory where you have write access. Multiple installations of WebSphere Application Server products or components require multiple locations.

app_client_root
The default installation root directory for the Java EE WebSphere Application Client is the /QIBM/ProdData/WebSphere/AppClient/V7/client directory.

app_client_user_data_root
The default Java EE WebSphere Application Client user data root is the /QIBM/UserData/WebSphere/AppClient/V7/client directory.

app_client_profile_root
The default Java EE WebSphere Application Client profile root is the /QIBM/UserData/WebSphere/AppClient/V7/client/profiles/*profile_name* directory.

app_server_root
The default installation root directory for WebSphere Application Server Network Deployment is the /QIBM/ProdData/WebSphere/AppServer/V7/ND/QIBM/ProdData/WebSphere/AppServer/V7/*product* directory.

cip_app_server_root
The default installation root directory is the /QIBM/ProdData/WebSphere/AppServer/V7/ND/QIBM/ProdData/WebSphere/AppServer/V7/*product* directory for a customized installation package (CIP) produced by the Installation Factory.

A CIP is a WebSphere Application Server Network Deployment product bundled with optional maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts.

cip_profile_root
The default profile root directory is the /QIBM/UserData/WebSphere/AppServer/V7/ND/cip/*cip_uid*/profiles/*profile_name*/QIBM/UserData/WebSphere/AppServer/V7/*product*/cip/*cip_uid*/profiles/*profile_name* directory for a customized installation package (CIP) produced by the Installation Factory.

cip_user_data_root
The default user data root directory is the /QIBM/UserData/WebSphere/AppServer/V7/ND/cip/*cip_uid*/QIBM/UserData/WebSphere/AppServer/V7/*product*/cip/*cip_uid* directory for a customized installation package (CIP) produced by the Installation Factory.

if_root This directory represents the root directory of the IBM WebSphere Installation Factory. Because you can download and unpack the Installation Factory to any directory on the file system to which you have write access, this directory's location varies by user. The Installation Factory is an Eclipse-based tool which creates installation packages for installing WebSphere Application Server in a reliable and repeatable way, tailored to your specific needs.

iip_root
This directory represents the root directory of an *integrated installation package* (IIP) produced by the IBM WebSphere Installation Factory. Because you can create and save an IIP to any directory on the file system to which you have write access, this directory's location varies by user. An IIP is an aggregated installation package created with the Installation Factory that can include one or more generally available installation packages, one or more customized installation packages (CIPs), and other user-specified files and directories.

java_home

Table 42. Root directories for supported Java Virtual Machines.

This table shows the root directories for all supported Java Virtual Machines (JVMs).

JVM	Directory
Classic JVM	/QIBM/ProdData/Java400/jdk6
32–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/32bit
64–bit IBM Technology for Java	/QOpenSys/QIBM/ProdData/JavaVM/jdk60/64bit

plugins_profile_root
: The default Web server plug-ins profile root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver/profiles/*profile_name* directory.

plugins_root
: The default installation root directory for Web server plug-ins is the /QIBM/ProdData/WebSphere/Plugins/V7/webserver directory.

plugins_user_data_root
: The default Web server plug-ins user data root is the /QIBM/UserData/WebSphere/Plugins/V7/webserver directory.

product_library
product_lib
: This is the product library for the installed product. The product library for each Version 7.0 installation on the system contains the program and service program objects (similar to .exe, .dll, .so objects) for the installed product. The product library name is QWAS7*x* (where *x* is A, B, C, and so on). The product library for the first WebSphere Application Server Version 7.0 product installed on the system is QWAS7A. The *app_server_root*/properties/product.properties file contains the value for the product library of the installation, was.install.library, and is located under the *app_server_root* directory.

profile_root
: The default directory for a profile named *profile_name* for WebSphere Application Server Network Deployment is the /QIBM/UserData/WebSphere/AppServer/V7/ND/profiles/*profile_name*/QIBM/UserData/WebSphere/AppServer/V7/*product*/profiles/*profile_name* directory.

shared_product_library
: The shared product library, which contains all of the objects shared by all installations on the system, is QWAS7. This library contains objects such as the product definition, the subsystem description, the job description, and the job queue.

updi_root
: The default installation root directory for the Update Installer for WebSphere Software is the /QIBM/ProdData/WebSphere/UpdateInstaller/V7/updi directory.

user_data_root
: The default user data directory for WebSphere Application Server Network Deployment is the /QIBM/UserData/WebSphere/AppServer/V7/ND/QIBM/UserData/WebSphere/AppServer/V7/*product* directory.

  The profiles and profileRegistry subdirectories are created under this directory when you install the product.

web_server_root
: The default web server path is /www/*web_server_name*.

Default product locations (distributed)

The following file paths are default locations. You can install the product and other components or create profiles in any directory where you have write access. Multiple installations of WebSphere Application Server Network Deployment products or components require multiple locations. Default values for installation actions by root and non-root users are given. If no non-root values are specified, then the default directory values are applicable to both root and non-root users.

app_client_root

Table 43. Default installation root directories for the WebSphere Application Client.

This table shows the default installation root directories for the WebSphere Application Client.

User	Directory
Root	/usr/IBM/WebSphere/AppClient (Java EE Application client only) /opt/IBM/WebSphere/AppClient (Java EE Application client only) C:\Program Files\IBM\WebSphere\AppClient
Non-root	*user_home*/IBM/WebSphere/AppClient (Java EE Application client only) C:\IBM\WebSphere\AppClient

app_server_root

Table 44. Default installation directories for WebSphere Application Server.

This table shows the default installation directories for WebSphere Application Server Network Deployment.

User	Directory
Root	/usr/IBM/WebSphere/AppServer /opt/IBM/WebSphere/AppServer C:\Program Files\IBM\WebSphere\AppServer
Non-root	*user_home*/IBM/WebSphere/AppServer C:\IBM\WebSphere\AppServer

cip_app_server_root

A *customized installation package* (CIP) is an installation package created with IBM WebSphere Installation Factory that contains a WebSphere Application Server or feature pack product bundled with one or more maintenance packages, an optional configuration archive, one or more optional enterprise archive files, and other optional files and scripts.

Table 45. Default installation root directories for a CIP.

This table shows the default installation root directories for a CIP.

User	Directory
Root	/usr/IBM/WebSphere/AppServer /opt/IBM/WebSphere/AppServer C:\Program Files\IBM\WebSphere\AppServer
Non-root	*user_home*/IBM/WebSphere/AppServer C:\IBM\WebSphere\AppServer

component_root

The component installation root directory is any installation root directory described in this topic. Some programs are for use across multiple components. In particular, the Update Installer for WebSphere Software is for use with WebSphere Application Server Network Deployment, Web server plug-ins, the Application Client, and the IBM HTTP Server. All of these components are part of the product package.

gskit_root

IBM Global Security Kit (GSKit) can now be installed by any user. GSKit is installed locally inside the installing product's directory structure and is no longer installed in a global location on the target system. The following list shows the default installation root directory for Version 7 of the GSKit, where *product_root* is the root directory of the product that is installing GSKit, for example IBM HTTP Server or the Web server plug-in.

UNIX

product_root/gsk7

Windows

product_root\gsk7

if_root This directory represents the root directory of the IBM WebSphere Installation Factory. Because you can download and unpack the Installation Factory to any directory on the file system to which you have write access, this directory's location varies by user. IBM WebSphere Installation Factory is an Eclipse-based tool which creates installation packages for installing WebSphere Application Server in a reliable and repeatable way, tailored to your specific needs.

iip_root

This directory represents the root directory of an *integrated installation package* (IIP) produced by the IBM WebSphere Installation Factory. Because you can create and save an IIP to any directory on the file system to which you have write access, this directory's location varies by user. An IIP is an aggregated installation package that can include one or more generally available installation packages, one or more customized installation packages (CIPs), and other user-specified files and directories.

profile_root

Table 46. Default profile directories.

This table shows the default directories for a profile named *profile_name* on each distributed operating system.

User	Directory
Root	AIX /usr/IBM/WebSphere/AppServer/profiles/*profile_name* HP-UX Linux Solaris /opt/IBM/WebSphere/AppServer/profiles/*profile_name* Windows C:\Program Files\IBM\WebSphere\AppServer\profiles*profile_name*
Non-root	HP-UX Linux Solaris *user_home*/IBM/WebSphere/AppServer/profiles/ Windows C:\IBM\WebSphere\AppServer\profiles\

plugins_root

Table 47. Default installation root directories for the Web server plug-ins.

This table shows the default installation root directories for the Web server plug-ins for WebSphere Application Server.

User	Directory
Root	. /usr/IBM/WebSphere/Plugins . . . /opt/IBM/WebSphere/Plugins . C:\Program Files\IBM\WebSphere\Plugins
Non-root	. . . . *user_home*/IBM/WebSphere/Plugins . C:\IBM\WebSphere\Plugins

Note: If the Web server plug-ins are installed as part of the IBM HTTP Server installation, the installation location is inside the IBM HTTP Server installation location. For example:

```
/opt/IBM/HTTPServer/Plugins
```

updi_root

Table 48. Default installation root directories for the Update Installer for WebSphere Software.

This table shows the default installation root directories for the Update Installer for WebSphere Software.

User	Directory
Root	. /usr/IBM/WebSphere/UpdateInstaller . . . /opt/IBM/WebSphere/UpdateInstaller . C:\Program Files\IBM\WebSphere\UpdateInstaller
Non-root	. . . . *user_home*/IBM/WebSphere/UpdateInstaller . C:\IBM\WebSphere\UpdateInstaller

web_server_root

Table 49. Default installation root directories for the IBM HTTP Server.

This table shows the default installation root directories for the IBM HTTP Server.

User	Directory
Root	. /usr/IBM/HTTPServer . . . /opt/IBM/HTTPServer . C:\Program Files\IBM\HTTPServer
Non-root	. . . . *user_home*/IBM/HTTPServer . C:\IBM\HTTPServer

Appendix A. 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 the following address:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594
USA

Appendix B. Trademarks and service marks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. For a current list of IBM trademarks, visit the IBM Copyright and trademark information Web site (www.ibm.com/legal/copytrade.shtml).

Microsoft and Windows 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.

Java is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.