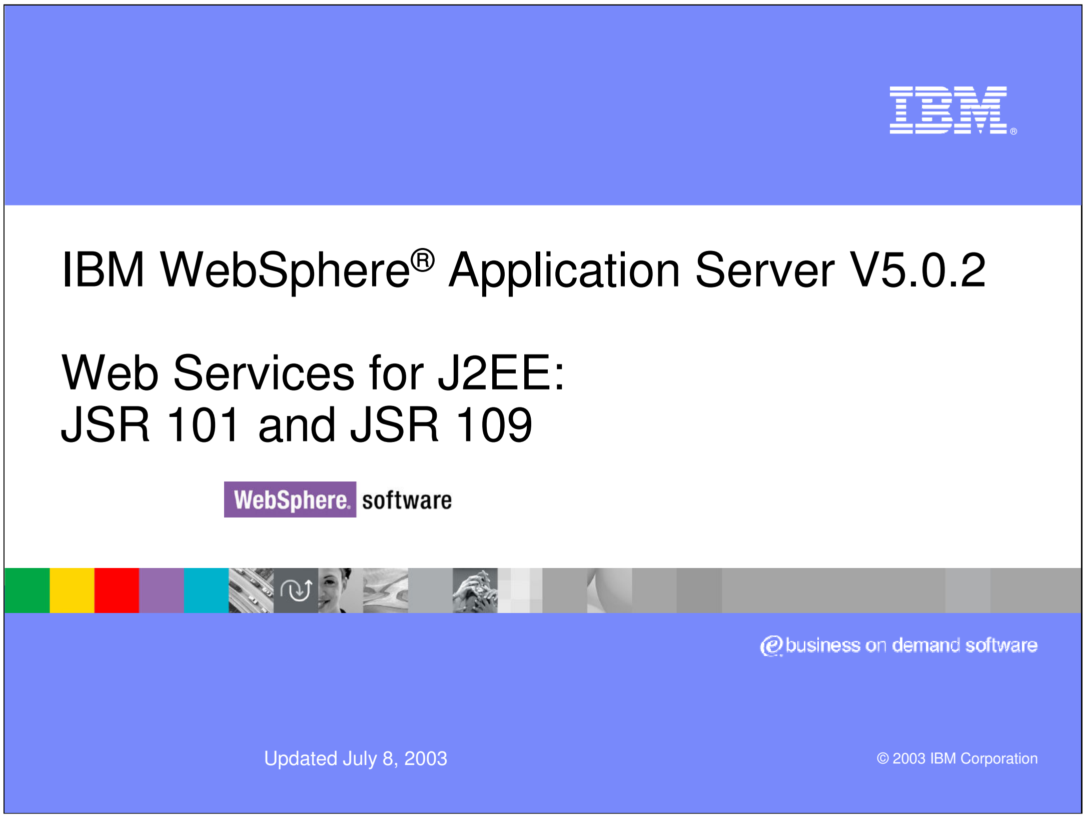


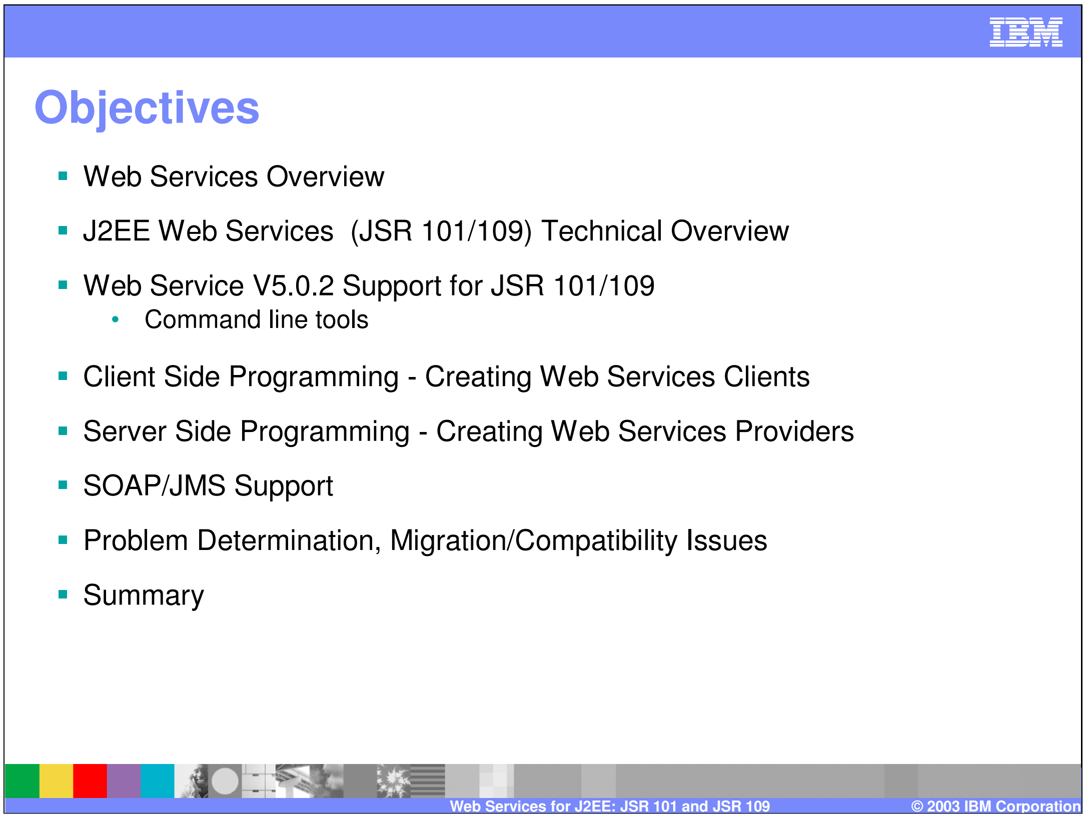
IBM WebSphere® Application Server V5.0.2

Web Services for J2EE: JSR 101 and JSR 109

WebSphere. software

@business on demand software

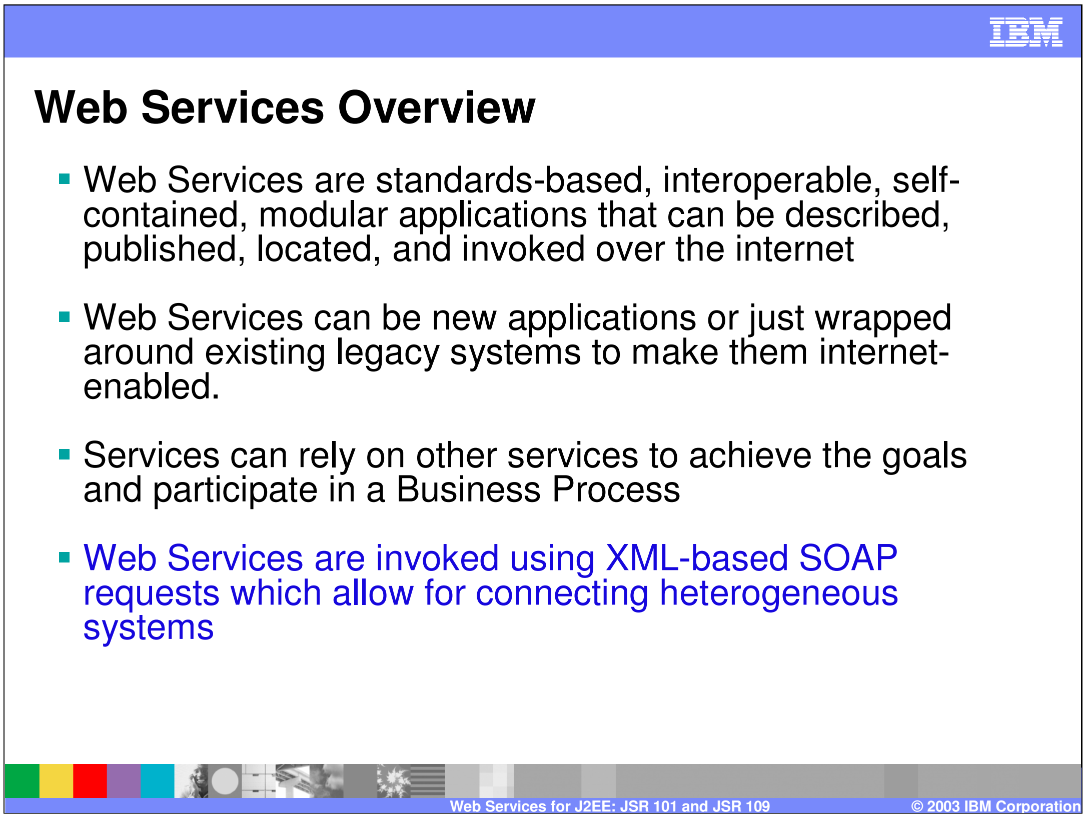
Updated July 8, 2003



Objectives

- Web Services Overview
- J2EE Web Services (JSR 101/109) Technical Overview
- Web Service V5.0.2 Support for JSR 101/109
  - Command line tools
- Client Side Programming - Creating Web Services Clients
- Server Side Programming - Creating Web Services Providers
- SOAP/JMS Support
- Problem Determination, Migration/Compatibility Issues
- Summary

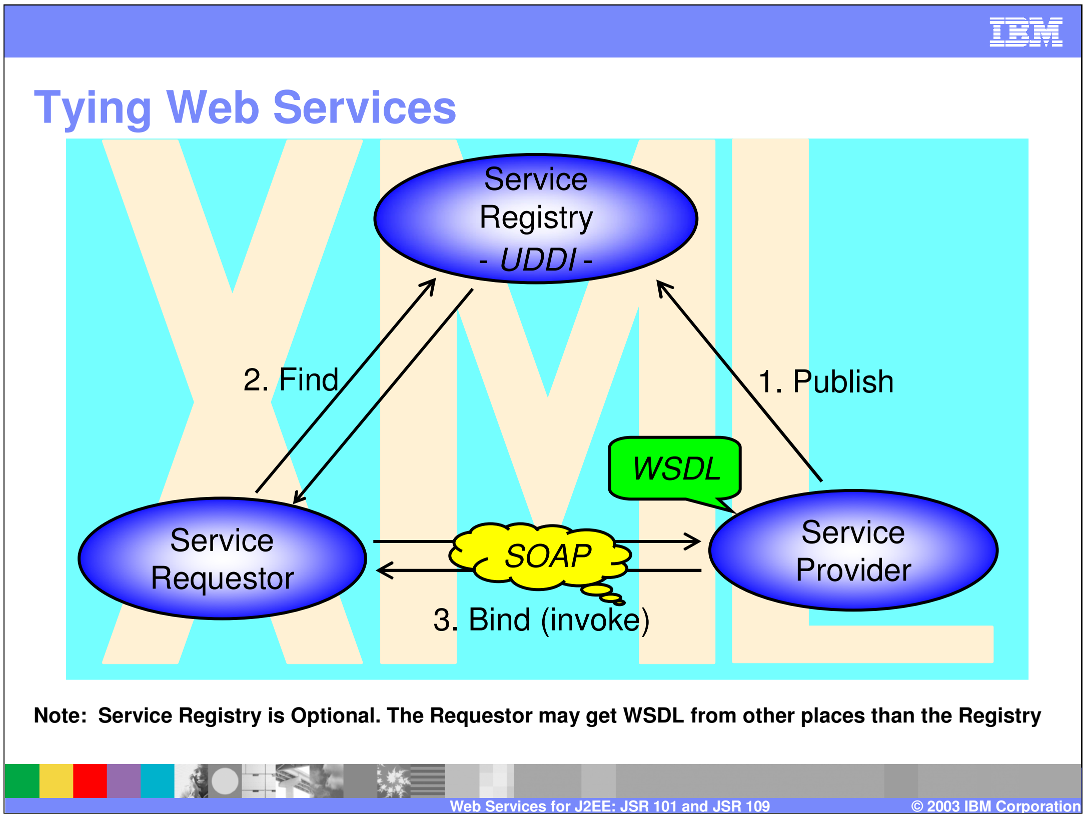
Web Services for J2EE: JSR 101 and JSR 109



Web Services Overview

- Web Services are standards-based, interoperable, self-contained, modular applications that can be described, published, located, and invoked over the internet
- Web Services can be new applications or just wrapped around existing legacy systems to make them internet-enabled.
- Services can rely on other services to achieve the goals and participate in a Business Process
- Web Services are invoked using XML-based SOAP requests which allow for connecting heterogeneous systems

This chart gives you a brief introduction on the salient Web Services concepts.



Tying Web Services

Note: Service Registry is Optional. The Requestor may get WSDL from other places than the Registry

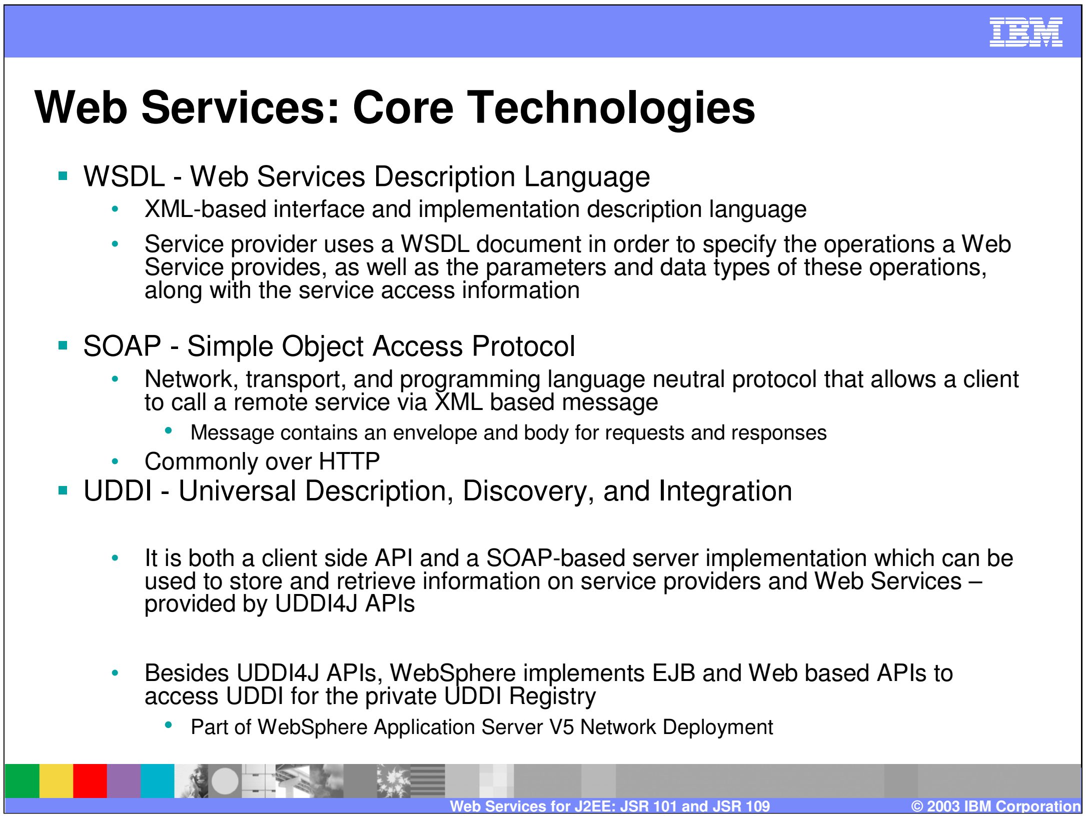
Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

This slide is the big picture explanation of web services. As you can see there are three roles in the triangle: the Service Provider, the UDDI registry, and the Service Requestor. Web Services are about interfaces, and primarily about a standard description of the services being provided. In particular, the WSDL document tells the client exactly where the service is being hosted, what are the operations that can be called, what are the input parameters, and what to expect back as a response from the service provider.

The service provider can exist on the internet or an intranet and it may choose to publish its services on a registry called the UDDI registry. This can be a public registry that anybody can access, or a private UDDI registry, accessible from within the Intranet.

Ultimately – whether through a UDDI registry or via other means, the service requester is interested in getting hold of the WSDL document that describes the service to be accessed. Using the WSDL document, the requester can generate a client that consumes the service.



Web Services: Core Technologies

- WSDL - Web Services Description Language
  - XML-based interface and implementation description language
  - Service provider uses a WSDL document in order to specify the operations a Web Service provides, as well as the parameters and data types of these operations, along with the service access information
- SOAP - Simple Object Access Protocol
  - Network, transport, and programming language neutral protocol that allows a client to call a remote service via XML based message
    - Message contains an envelope and body for requests and responses
  - Commonly over HTTP
- UDDI - Universal Description, Discovery, and Integration
  - It is both a client side API and a SOAP-based server implementation which can be used to store and retrieve information on service providers and Web Services – provided by UDDI4J APIs
  - Besides UDDI4J APIs, WebSphere implements EJB and Web based APIs to access UDDI for the private UDDI Registry
    - Part of WebSphere Application Server V5 Network Deployment

Web Services for J2EE: JSR 101 and JSR 109

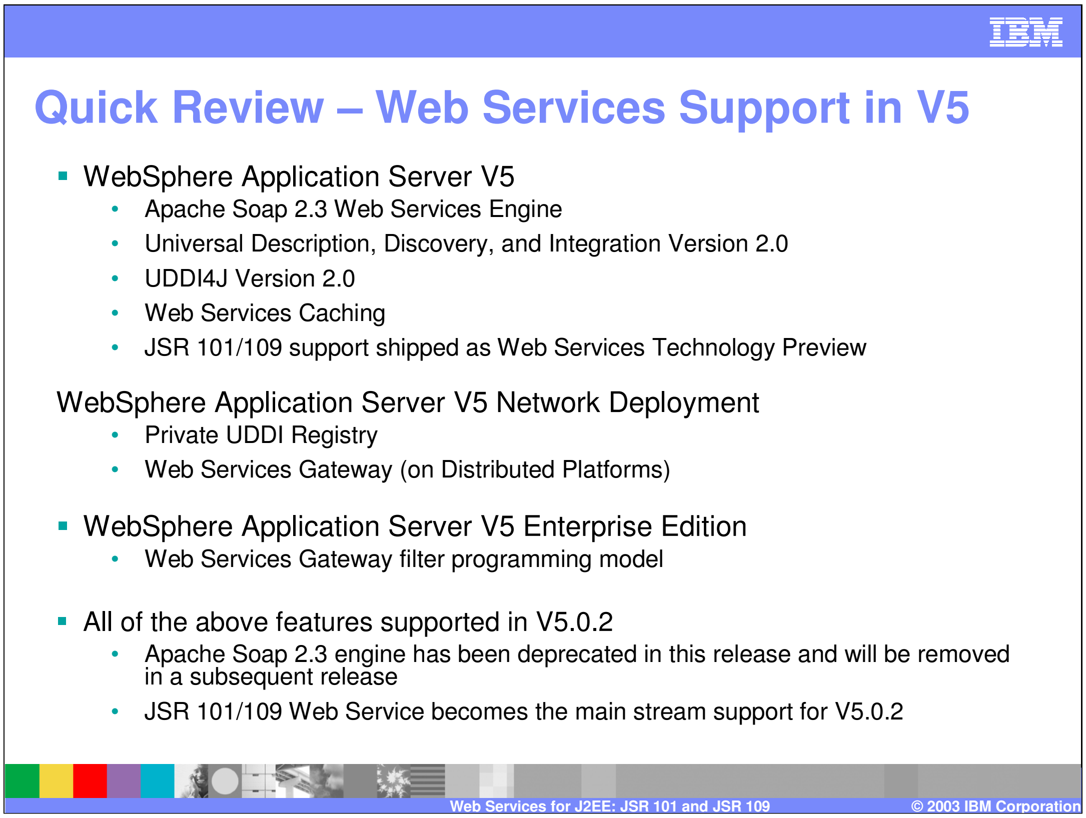
WSDL is nothing more than an XML based interface which describes:

1) The location of the service (or service definition) – end point location
2) The interface of your service
3) The bindings to the service, in terms of data mapping and translation.

The interaction between the requester and the provider occurs by means of an invocation protocol called SOAP or Simple Object Access Protocol. SOAP carries an XML document that goes from the requester to the provider, and a SOAP response would contain an XML formulation of the response from the provider.

Web Services are commonly implemented using SOAP over HTTP, but they can also be implemented using SOAP over JMS.

The UDDI registry is an option that allows the provide to “advertise” its services, and the requester to “discover” the most interesting services. If you have a WSDL document and you want to publish it so that a client can get access to it dynamically and call your web service – the UDDI registry is one way to achieve that. The registry can be accessed in a variety of ways, including some special Java APIs called UDDI4J which stands for UDDI for Java. These APIs allow both publishing and retrieving a service.



Quick Review – Web Services Support in V5

- WebSphere Application Server V5
  - Apache Soap 2.3 Web Services Engine
  - Universal Description, Discovery, and Integration Version 2.0
  - UDDI4J Version 2.0
  - Web Services Caching
  - JSR 101/109 support shipped as Web Services Technology Preview

WebSphere Application Server V5 Network Deployment

  - Private UDDI Registry
  - Web Services Gateway (on Distributed Platforms)

- WebSphere Application Server V5 Enterprise Edition
  - Web Services Gateway filter programming model

- All of the above features supported in V5.0.2
  - Apache Soap 2.3 engine has been deprecated in this release and will be removed in a subsequent release
  - JSR 101/109 Web Service becomes the main stream support for V5.0.2

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

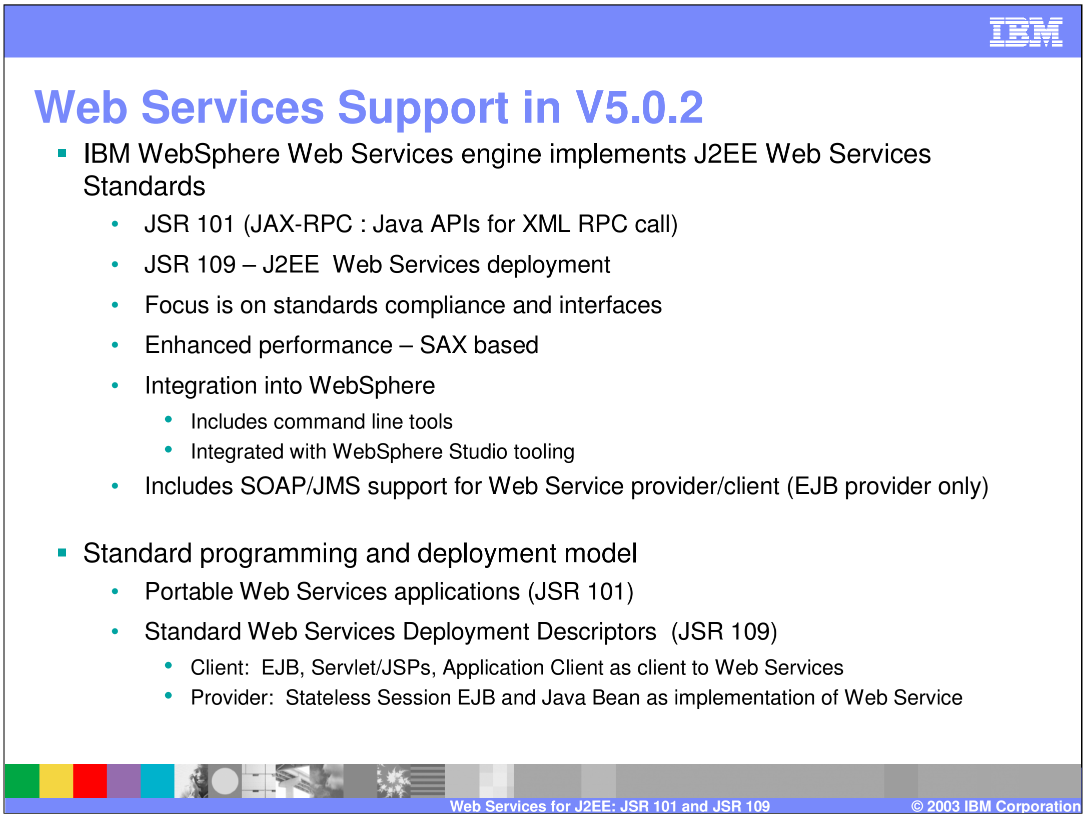
Between V5 and V5.0.2 the JSR 101 and JSR 109 specifications have solidified. WS-Security is also supported in production.

In this module, the focus will be on JSR 101 and JSR 109.

The Web Services Gateway is available on the distributed platforms and is not available on the z/OS platform.

The WebSphere application Server V5 Enterprise enhances the Web Services Gateway to include the ability to write custom filters. WebSphere Application Server Network Deployment had some filtering functions, but there was no way for you to create your own custom filters.

Currently the old Web Services Gateway that was based on the Apache SOAP engine is getting transformed to be based on the new J2EE Web Services engine. The Apache SOAP engine will continue to be supported in WebSphere Application Server V5.0.2



Web Services Support in V5.0.2

- IBM WebSphere Web Services engine implements J2EE Web Services Standards
  - JSR 101 (JAX-RPC : Java APIs for XML RPC call)
  - JSR 109 – J2EE Web Services deployment
  - Focus is on standards compliance and interfaces
  - Enhanced performance – SAX based
  - Integration into WebSphere
    - Includes command line tools
    - Integrated with WebSphere Studio tooling
  - Includes SOAP/JMS support for Web Service provider/client (EJB provider only)
- Standard programming and deployment model
  - Portable Web Services applications (JSR 101)
  - Standard Web Services Deployment Descriptors (JSR 109)
    - Client: EJB, Servlet/JSPs, Application Client as client to Web Services
    - Provider: Stateless Session EJB and Java Bean as implementation of Web Service

Web Services for J2EE: JSR 101 and JSR 109

The web services support in WebSphere 5.0.2 is called the IBM Web Services for J2EE engine which is purely based on the JSR 101/109 specifications. The new engine also includes the new SAX parser which is predominantly faster than the DOM parser that the Apache SOAP engine was based on.

Command line tooling is provided with WebSphere, to generate most artifacts for the JSR 101/109 compliance and to facilitate Web Services development and deployment. The Application Server Toolkit also includes tooling to facilitate the assembly. Additional tooling will be made available shortly within the WebSphere Studio product.

The focus of the IBM Web Services engine in V5.0.2 is to be compliant with a standard programming and deployment model.

If you look at the Apache SOAP engine the standardization was achieved at the SOAP level. There was no standardization in terms of the programming model (for consuming the service) or deployment model (for installing the service on the runtime). That meant that a service created for V5 would only run on WebSphere Application Server. Also, a client that used the Apache SOAP support to consume the service, would also not be portable.

Now with JSR 101/109, there is a standard programming model: when you create a client, that client will work independently of which application server is on the other side. Also, when you create your web service and deploy it, the same deployment model holds true for different application servers.

This is achieved by having additional deployment descriptors to support the standard deployment model – and this is the focus of JSR 109.

JSR 101 gives us the “underpinnings” – the standard APIs that can be used to invoke a service in a standard fashion. JSR 109 is more focused on the deployment model or in other words, how a J2EE application which is Web Services enabled can be installed and deployed on any compliant application server.

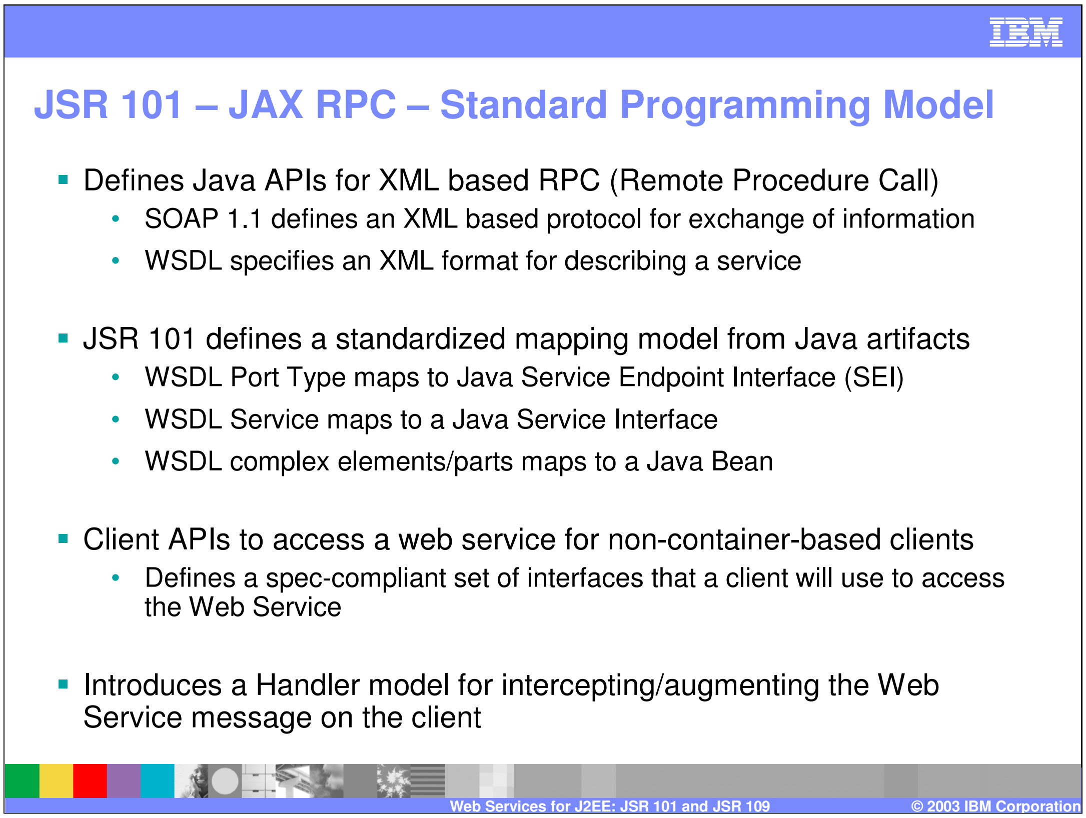
Summarizing:

The SOAP based WebSphere Application Server V5 Web Services in V5 is

Apache SOAP 2.3 runtime
DOM-based (slow)
Proprietary programming model

WebSphere 5.0.2 supports:

•Simple Object Access Protocol (SOAP) Version 1.1
•Web Services Description Language (WSDL) Version 1.1
•Web Services for J2EE (JSR-109) Version 1.0
•Java API for XML-Based RPC (JAX-RPC) Version 1.0
•SOAP with Attachments API for Java (SAAJ) Version 1.1



JSR 101 – JAX RPC – Standard Programming Model

- Defines Java APIs for XML based RPC (Remote Procedure Call)
  - SOAP 1.1 defines an XML based protocol for exchange of information
  - WSDL specifies an XML format for describing a service
- JSR 101 defines a standardized mapping model from Java artifacts
  - WSDL Port Type maps to Java Service Endpoint Interface (SEI)
  - WSDL Service maps to a Java Service Interface
  - WSDL complex elements/parts maps to a Java Bean
- Client APIs to access a web service for non-container-based clients
  - Defines a spec-compliant set of interfaces that a client will use to access the Web Service
- Introduces a Handler model for intercepting/augmenting the Web Service message on the client

Web Services for J2EE: JSR 101 and JSR 109

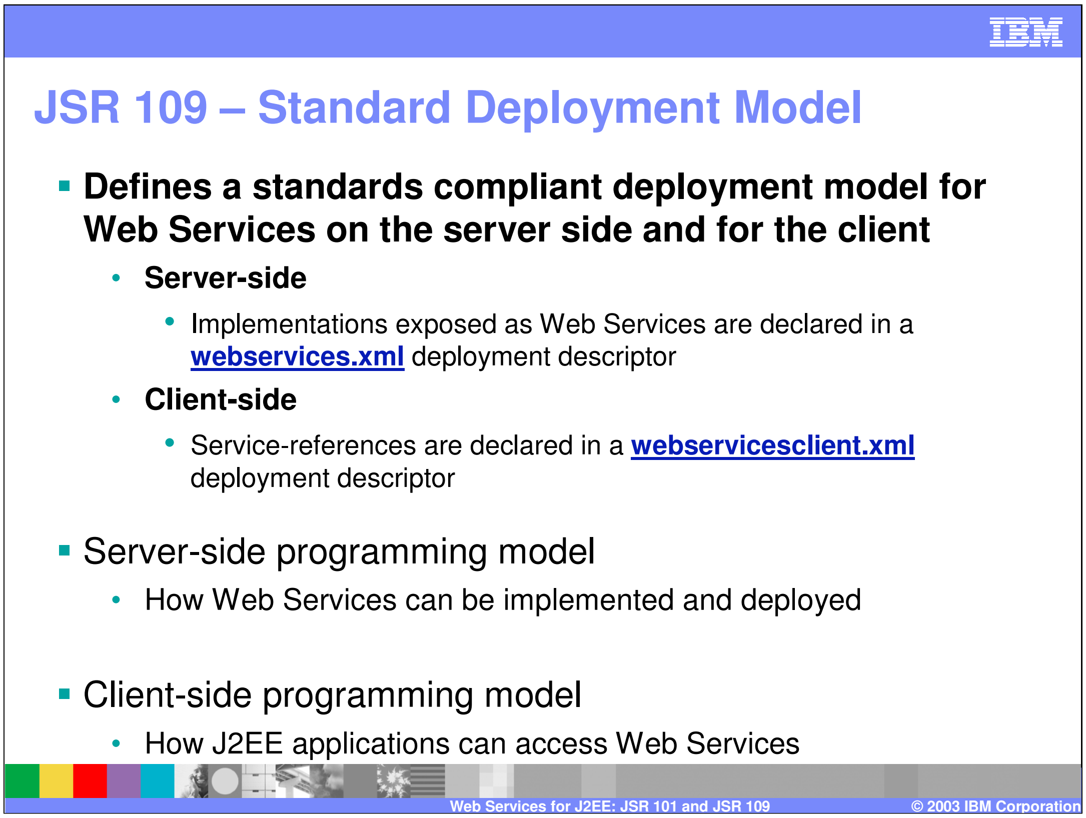
Let's focus on JSR 101 also known as JAX-RPC. This layer defines the basic programming model for Web Services invocation. These APIs assume that a WSDL document is available at the time the service is invoked. WSDL documents describe all aspects that permit a service invocation.

The JSR 101 defines the standardized mapping model between the WSDL and the Java artifacts.

For instance, a WSDL document defines a port type – which in turn includes a number of operations, and those operations exchange a series of messages. Messages are made of parts, or elemental data types. All these concepts need to be mapped into Java artifacts, in a standard way – and this is exactly what JSR 101 defines. For example, a Port Type will be mapped to a special Java interface, called the Service Endpoint Interface (SEI). Operations map to methods on the SEI. The messages and parts map to Java beans, when they do not map directly to primitive Java data types. The WSDL service definition (which identifies the endpoint URL) may map to a Java Service interface, optionally.

In addition, JSR 101 defines are the client APIs that allow a Java service requester to consume a certain service. A service requester has a number of options, which are discussed in detail later in this module. One of these options is to use the JSR 101 APIs directly – in this case, the client will not run within a J2EE container. More typically, a client will run in a "managed" mode, that is, within a J2EE container. In that case, it will use the JSR 109 programming model, which shields the Web Services client from the details of the JSR 101 APIs.

The JSR 101 also defines the concept of Handlers. Those are special Java classes that intercept requests and responses before they get to destination. They can do a number of things to those requests or responses, for instance, they can encrypt them, or audit them.



JSR 109 – Standard Deployment Model

- **Defines a standards compliant deployment model for Web Services on the server side and for the client**
  - **Server-side**
    - Implementations exposed as Web Services are declared in a **webservices.xml** deployment descriptor
  - **Client-side**
    - Service-references are declared in a **webservicesclient.xml** deployment descriptor
- Server-side programming model
  - How Web Services can be implemented and deployed
- Client-side programming model
  - How J2EE applications can access Web Services

Web Services for J2EE: JSR 101 and JSR 109

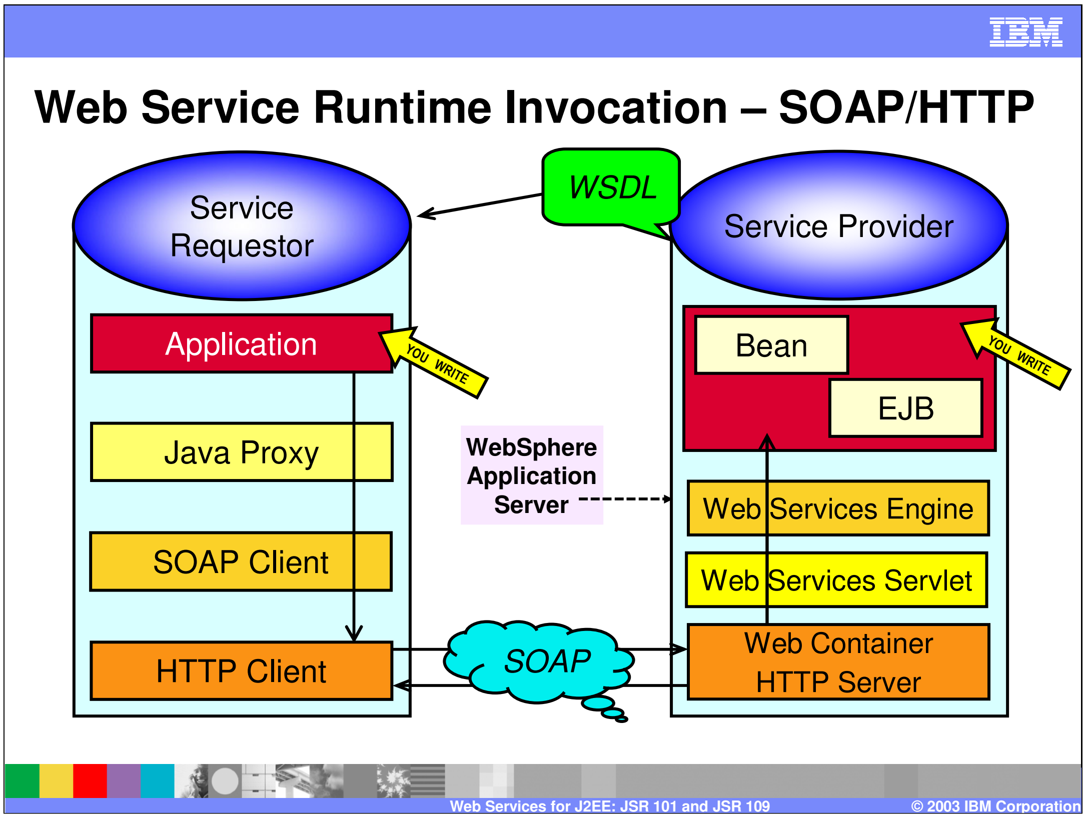
© 2003 IBM Corporation

Thus far, JSR 101 which defines the basic programming model, has been discussed. However, this JSR does not describe how to deploy those applications from a J2EE enterprise applications point of view so that you have commonalities among different application server vendors. In Version 5.0.2, JSR 109 is fully supported, and this JSR defines a standard deployment model for Web Services. Given a web service provider which is a Java bean or an EJB, this body of standard APIs defines standard deployment descriptors that describe those web services to an application server. JSR 109 also defines special deployment descriptor for client applications, so that they can look up and invoke a Web Service that is deployed on the J2EE application server. The programming model is kept consistent, as far as possible, with the EJB programming model, where applications can look up an installed EJB and invoke it. Similarly, Web Services requesters would be capable of looking up and invoking a deployed Web Service.

On the server side, JSR 109 defines an xml file called webservices.xml similar to the eib-jar.xml that defines the deployment descriptors for the EJB modules, and similar to the web.xml that defines the deployment descriptors for the war modules. The webservices.xml file needs to be packaged with the WAR file or EJB JAR file that contains the Web Services implementation (a Java bean, or an EJB).

On the client side, the J2EE application client will now have a webservicesclient.xml. This file contains the references to Web Services that are deployed on a J2EE application server.

So with JSR 101 and 109, there is a standard programming and deployment model – these JSRs will be integral part of the J2EE 1.4 specification.



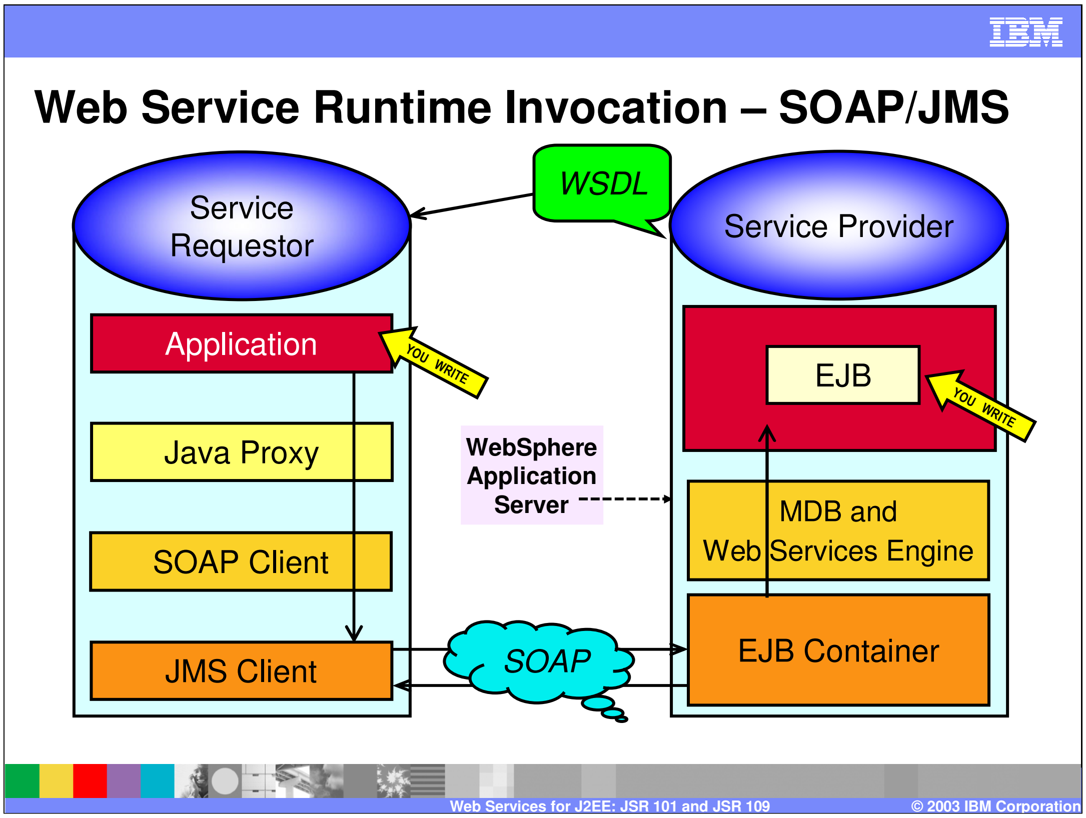
Web Service Runtime Invocation – SOAP/HTTP

Web Services for J2EE: JSR 101 and JSR 109

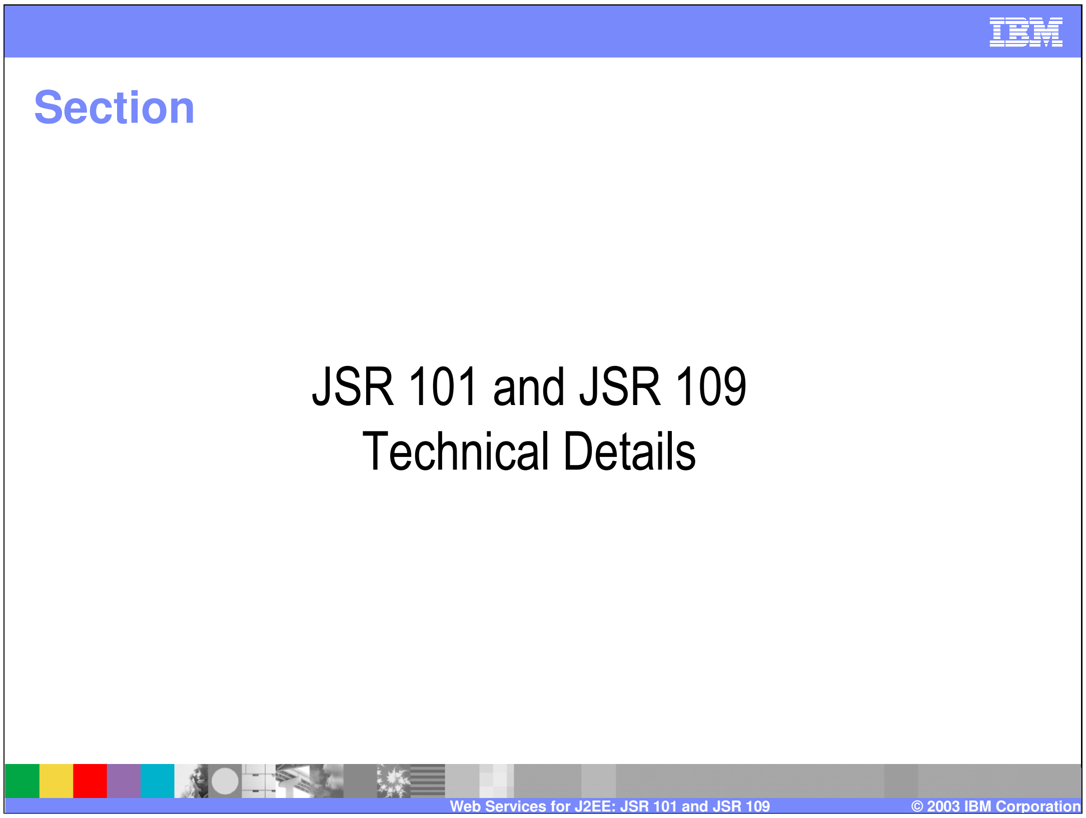
© 2003 IBM Corporation

At a very high level this page shows you the web services components that play a role when SOAP over HTTP is involved. Let's take a look at the provider: here you can see that you could have a Java bean or a stateless session EJB. These are the two Java artifacts that can be exposed by the web service provider. Then using command line tools or appropriate tooling, you can create e a WSDL document that describes the service implemented by that Java bean or enterprise bean.

The WSDL document is used by the client on the left hand side to be able to call the web service provider. When you are writing the application client code, you can use tooling to help you create the Java proxy that under the covers interacts with your SOAP client component. That component which implements the JAX-RPC programming model and sends the SOAP request over HTTP. Then on the provider side, the web container will receive the request through a servlet and call the web services engine that is implementing the JSR 101/109 specifications. Internally, the servlet will call the Java bean or enterprise bean.

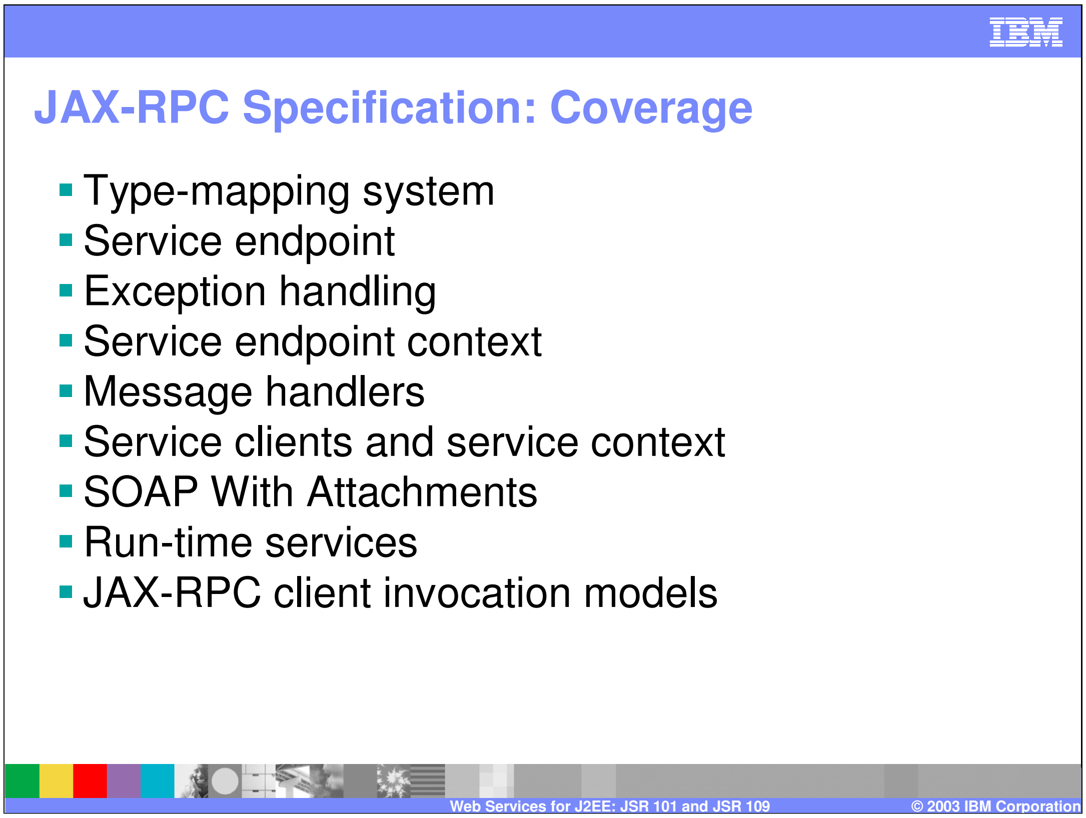


This chart describes the components and invocation steps involved in a SOAP/JMS interaction. These steps are very similar to those described in the previous chart, except that the SOAP client now generates, transparently, JMS calls, instead of issuing HTTP invocations. The end client will not realize the difference though – the proxy and the SOAP client shield the end programmer from the specifics of the protocol. The other difference on this chart is shown on the right-hand side – The Web services Engine is not implemented by a servlet any longer, but it is rather implemented by a Message-driven bean that listens on a specific destination where the SOAP messages are coming in.



Section

JSR 101 and JSR 109 Technical Details



JAX-RPC Specification: Coverage

- Type-mapping system
- Service endpoint
- Exception handling
- Service endpoint context
- Message handlers
- Service clients and service context
- SOAP With Attachments
- Run-time services
- JAX-RPC client invocation models

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

This chart shows you the coverage of the JAX-RPC specifications.

Type-mapping system - it’s the way WSDL messages, parts, and operations map to Java artifacts.

Service endpoint -- endpoint that actually defines your Java artifact (an HTTP URL or a JMS URL)

Exception handling – done by JAX RPC if you Java program needs to send back an exception

Service endpoint context – Ability to propagate context information (security, transaction, etc.)

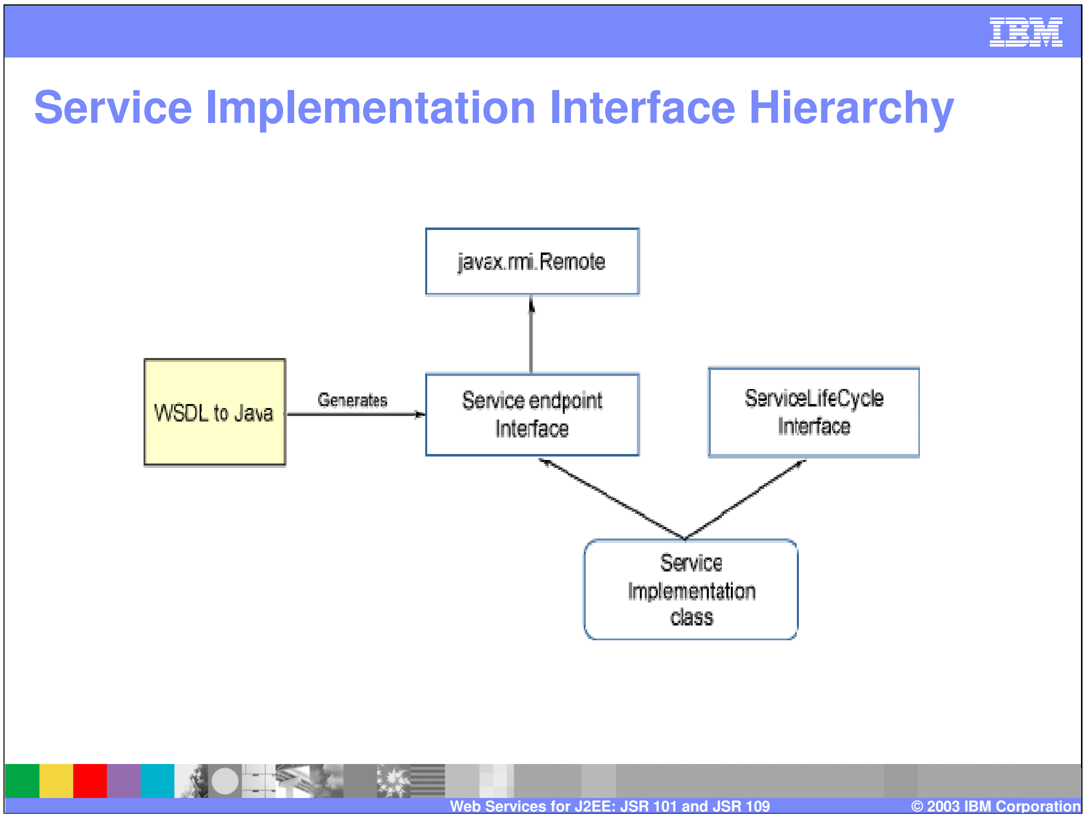
Message handlers – Defines specific APIs and a programming model to develop SOAP message handlers

Service clients and service context – how you create JAX RPC

SOAP With Attachments – a protocol that defines how MIME encoded parameters can be placed on a SOAP request or response

Run-time services -- should be able to take same services across different application server vendors

JAX-RPC client invocation models – there are two models to choose from. One is based on generating a stub and using it at runtime to perform the invocation. The second client invocation model is based on Dynamic Invocation Interface (DII), where the client builds the call dynamically, without the need of a precompiled stub.



Service Implementation Interface Hierarchy

javax.rmi.Remote
WSDL to Java
Generates
Service endpoint Interface
ServiceLifeCycle Interface
Service Implementation class

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

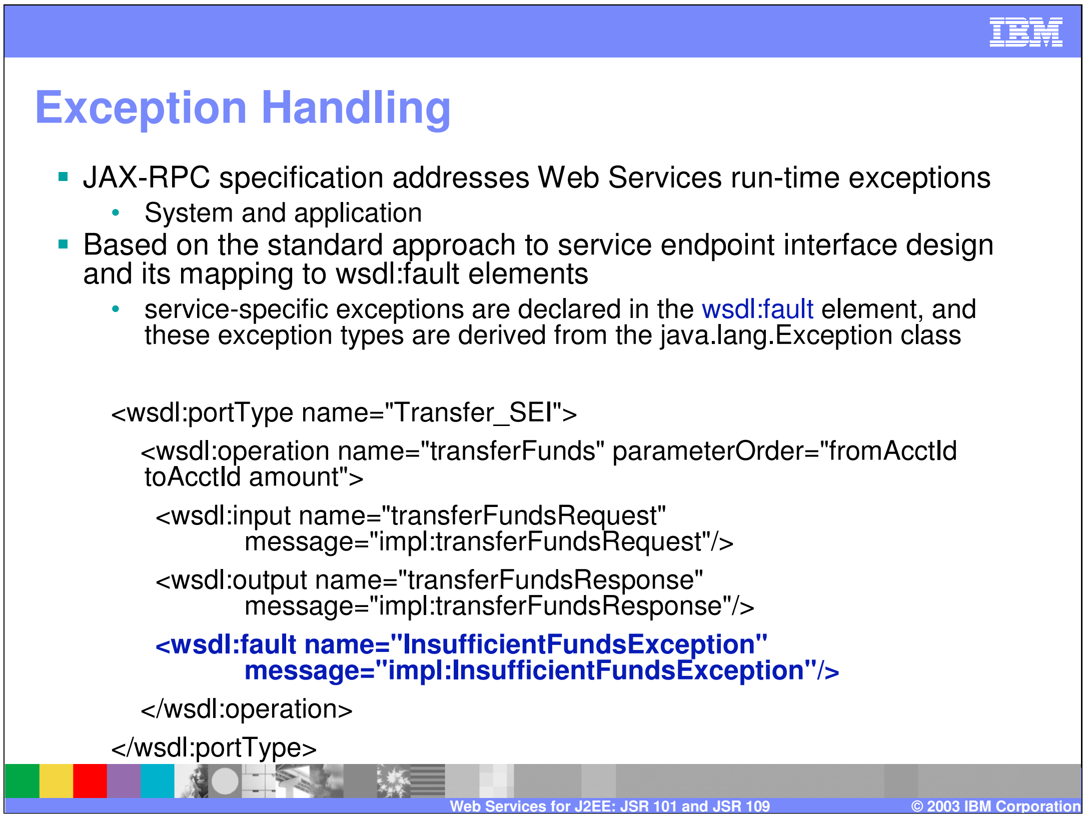
The Service Implementation class can optionally implement the ServiceLifeCycle Interface (shown below:)

```
Public interface ServiceLifecycle{
   void init (Object context) throws ServiceException;
  void destroy();
}
```

The runtime system is responsible for loading the service endpoint class and instantiating it. After instantiation, it invokes the ServiceLifecycle.init() method to initialize the service instance. ServiceLifecycle.init() expects the run-time context (ServiceContext) as a parameter.

The runtime system will invoke the ServiceLifecycle.destroy() method when the runtime system removes the Service endpoint from servicing the clients.

Service Endpoint should not maintain any state – The runtime may dispatch multiple client invocations to the service endpoint interface to this single instance.



Exception Handling

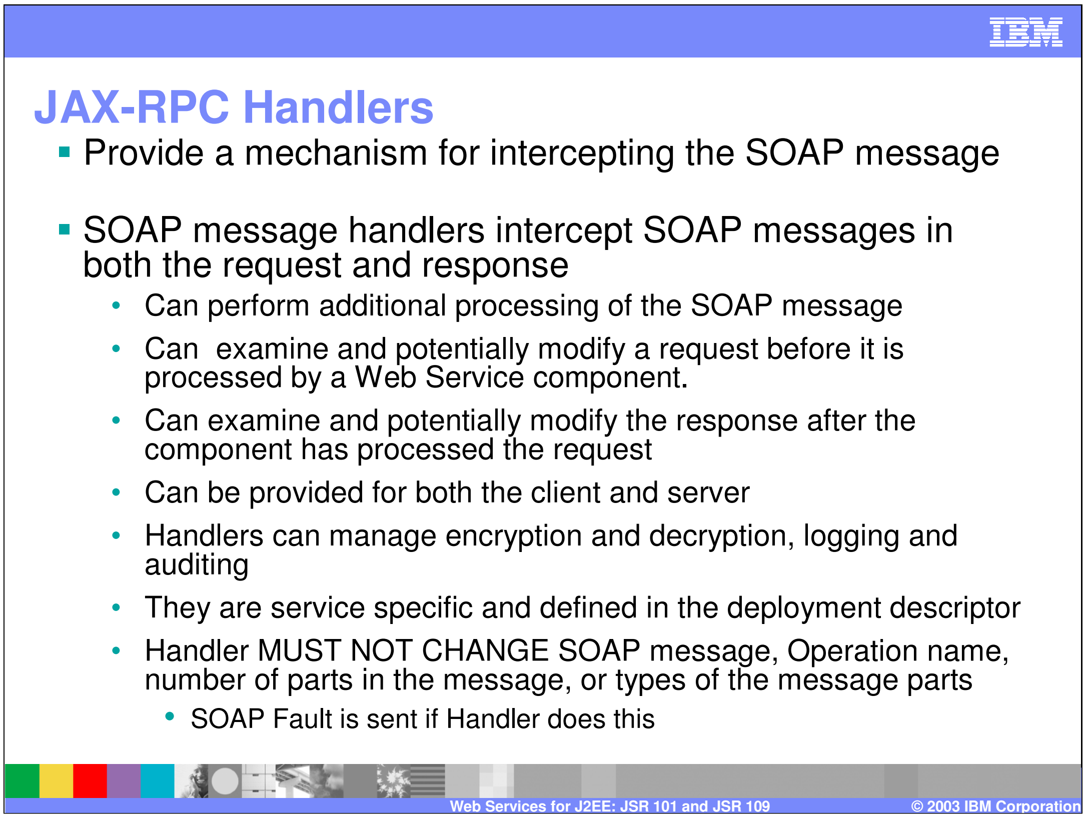
- JAX-RPC specification addresses Web Services run-time exceptions
  - System and application
- Based on the standard approach to service endpoint interface design and its mapping to wsdl:fault elements
  - service-specific exceptions are declared in the wsdl:fault element, and these exception types are derived from the java.lang.Exception class

```
<wsdl:portType name="Transfer_SEI">
   <wsdl:operation name="transferFunds" parameterOrder="fromAcctId
   toAcctId amount">
     <wsdl:input name="transferFundsRequest"
             message="impl:transferFundsRequest"/>
     <wsdl:output name="transferFundsResponse"
             message="impl:transferFundsResponse"/>
     <wsdl:fault name="InsufficientFundsException"
             message="impl:InsufficientFundsException"/>
   </wsdl:operation>
</wsdl:portType>
```

Web Services for J2EE: JSR 101 and JSR 109 © 2003 IBM Corporation

WSDL includes the definition of *fault* elements to take into account the events where exceptions are thrown. The WSDL document that defines the port type also includes the JAX-RPC support for mapping those faults elements to Exception classes to be thrown by the SEI methods. In the example on the chart, the operation is transferFunds has one input message (the transferFundsRequest) and an output message transferFundsResponse.

If you try to transfer funds and there’s not enough money in the fromAccount then the EJB will generate an InsufficientsFunds exception which will be defined as a WSDL fault within the WSDL document which you can see highlighted in blue. JAX-RPC requires that those faults be exposed as Java Exceptions in the SEI. It also requires appropriate serializers and deserializers to be created and included with the web-services-enabled applications, so that the information associated with the Exception can flow on SOAP message.



JAX-RPC Handlers

- Provide a mechanism for intercepting the SOAP message
- SOAP message handlers intercept SOAP messages in both the request and response
  - Can perform additional processing of the SOAP message
  - Can examine and potentially modify a request before it is processed by a Web Service component.
  - Can examine and potentially modify the response after the component has processed the request
  - Can be provided for both the client and server
  - Handlers can manage encryption and decryption, logging and auditing
  - They are service specific and defined in the deployment descriptor
  - Handler MUST NOT CHANGE SOAP message, Operation name, number of parts in the message, or types of the message parts
    - SOAP Fault is sent if Handler does this

Web Services for J2EE: JSR 101 and JSR 109

JAX-RPC handlers provide a way for intercepting the SOAP requests and responses before they actually get to destination.

There are some things the handlers can do and some they can't do:

•They can modify a request.

•They cannot change the SOAP message, or the web services engine will send back a web services SOAP fault.

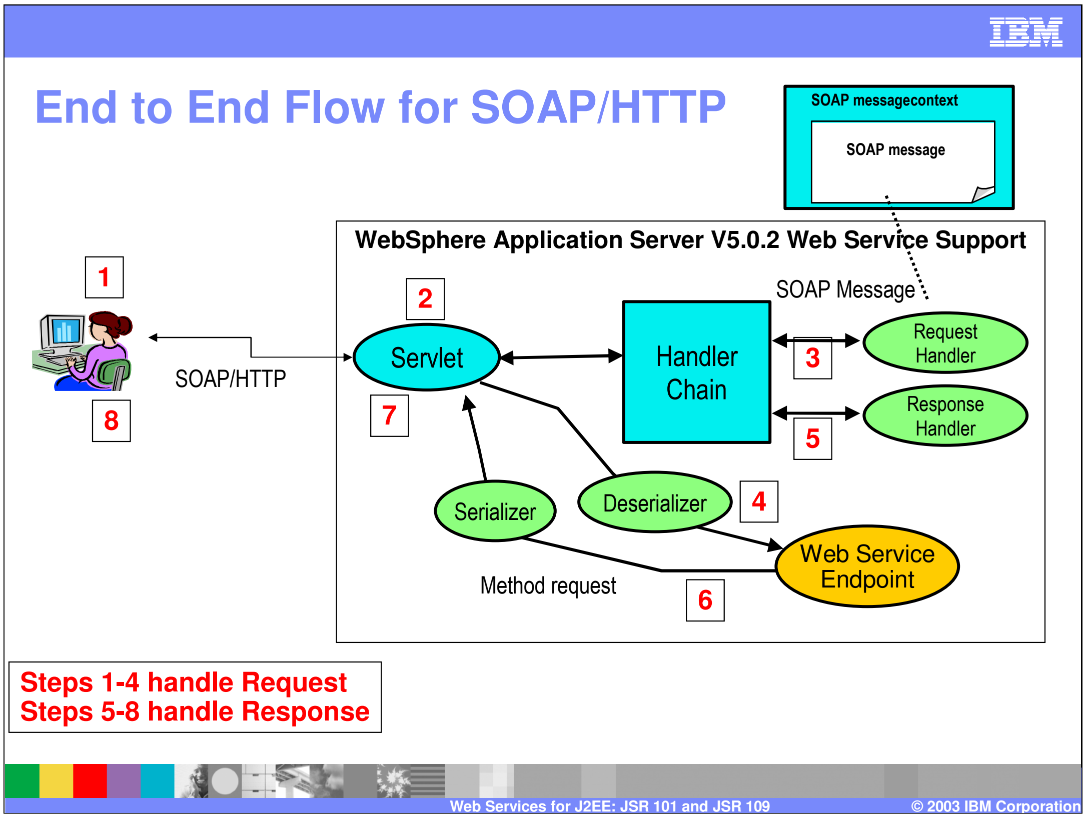
Handlers are very service specific so that you can define multiple handlers for specific services on the provider side and on client side.

Client side Handlers run after the Stub/proxy has marshaled the message, but before container services and the transport binding occurs.

Server side Handlers run after container services have run including method level authorization, but before demarshalling and dispatching the SOAP message to the endpoint.

There is no standard means for a Handler to access the security identity associated with a request, therefore Handlers cannot portably perform processing based on security identity.

A Handler must not change the message in any way that would cause the previously executed authorization check to execute differently.



End to End Flow for SOAP/HTTP

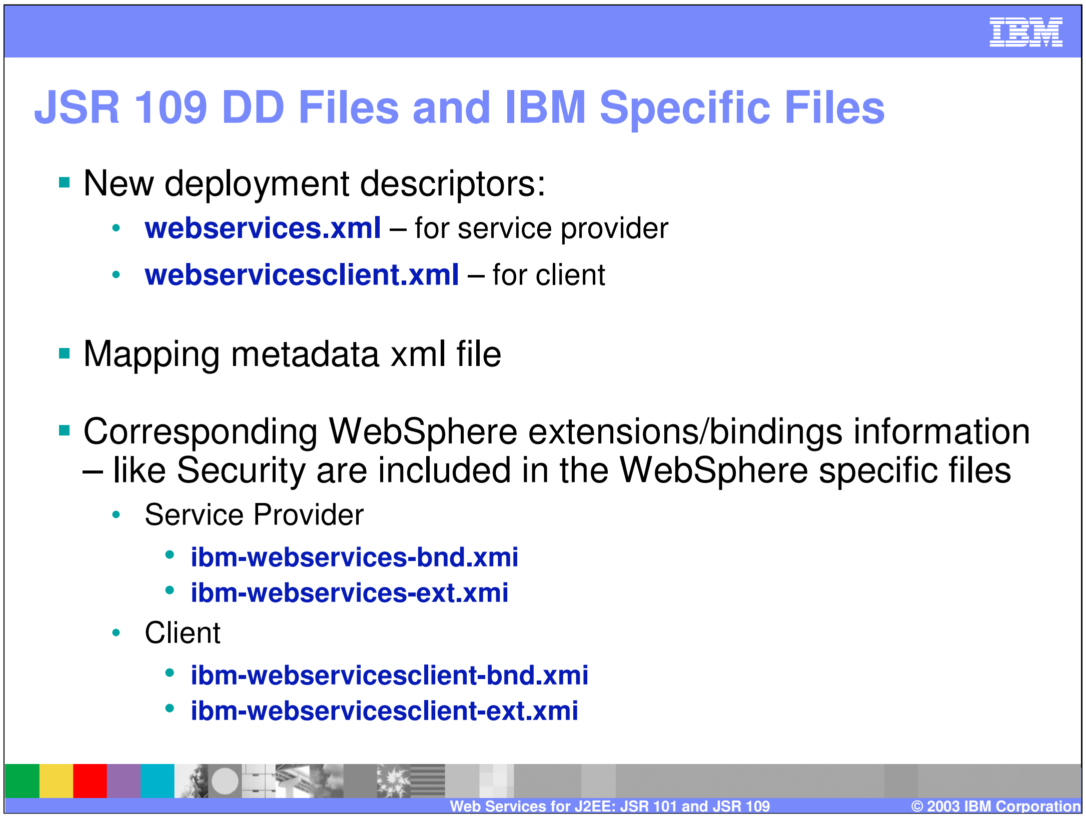
Steps 1-4 handle Request
Steps 5-8 handle Response

Web Services for J2EE: JSR 101 and JSR 109

In this example, the client is sending a SOAP over HTTP request, but this equally applies to SOAP over JMS, the only difference being that the intercepting point in the application server would be an Message-Driven bean instead of a servlet.

The servlet which is illustrated here is part of the web services engine – that's the servlet provided by WebSphere Application Server which processes the incoming SOAP requests. What the servlet receives is the entire SOAP envelope in the HTTP request. Then the web services engine will call the handlers that have been defined in the webservices.xml file which is the standard descriptor on the server side.

This picture shows you the server side - however a set of handlers can also be defined on the client side as well. Once the handlers are called then the deserializer happens when the WSDL2java mappings happens and then the web service endpoint is called. Once the response is produced, the web services engine will call the handlers in reverse order before the response goes back to the client.



JSR 109 DD Files and IBM Specific Files

- New deployment descriptors:
  - **webservices.xml** – for service provider
  - **webservicesclient.xml** – for client
- Mapping metadata xml file
- Corresponding WebSphere extensions/bindings information – like Security are included in the WebSphere specific files
  - Service Provider
    - **ibm-webservices-bnd.xmi**
    - **ibm-webservices-ext.xmi**
  - Client
    - **ibm-webservicesclient-bnd.xmi**
    - **ibm-webservicesclient-ext.xmi**

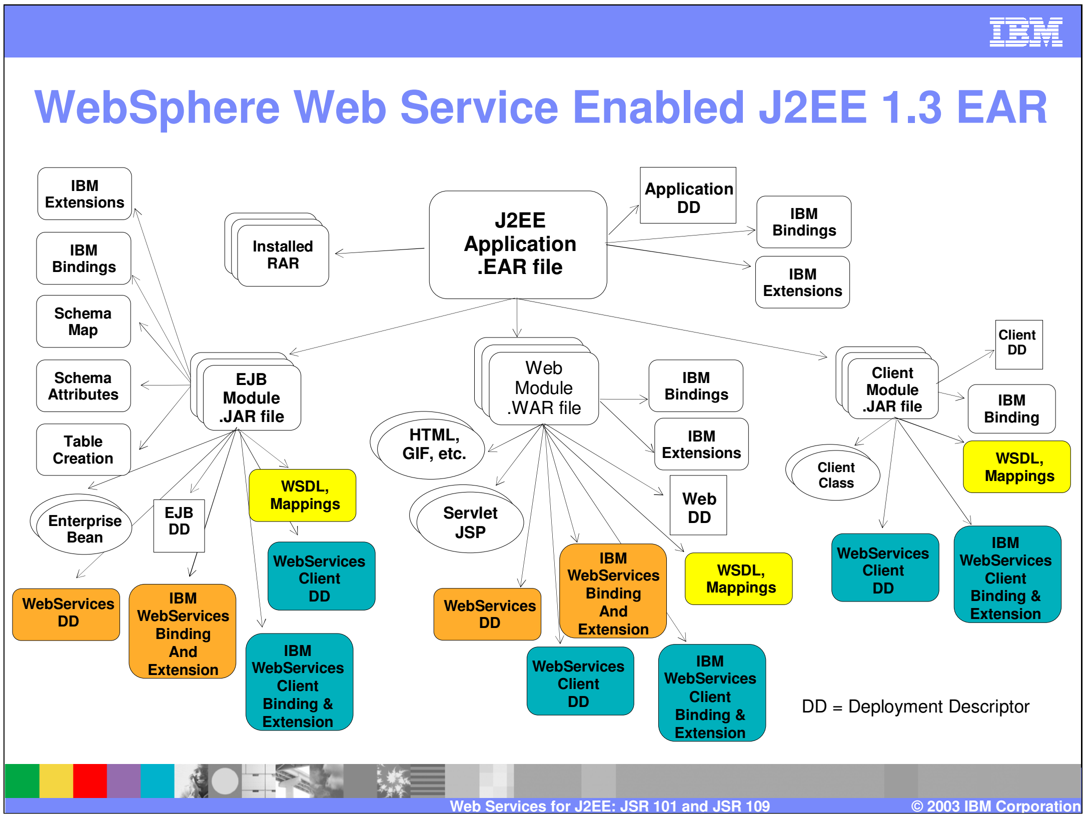
Web Services for J2EE: JSR 101 and JSR 109

As mentioned previously, a webservices.xml deployment descriptor file needs to be included within the EAR file that contains the service provider's implementation artifacts. In particular, you will include that file within a WAR file, if the service is implemented by a Java bean, or within an EJB JAR file, if the service is implemented by a stateless Session EJB.

A service requester application, whether it is a J2EE client, or a Web Application, or another EJB, needs to include the webservicesclient.xml deployment descriptor.

Both requester and provider applications need to include a mapping metadata xml file, which is generated by the tooling. This file is needed to map the complex xsd elements found in the WSDL file to the appropriate Java beans and vice versa

Just like in the EJB module there were IBM specific files, there are also IBM proprietary bindings and extensions for web services as well. These are aspects that are not defined within the specification but are required to run the EJB within the application server. These will be talked about more in the Web Services Security module because these files are where the WS-Security information is included.



WebSphere Web Service Enabled J2EE 1.3 EAR

J2EE Application .EAR file

Application DD

IBM Bindings

IBM Extensions

Installed RAR

EJB Module .JAR file

IBM Extensions

IBM Bindings

Schema Map

Schema Attributes

Table Creation

Enterprise Bean

EJB DD

WSDL, Mappings

WebServices Client DD

WebServices DD

IBM WebServices Binding And Extension

IBM WebServices Client Binding & Extension

Web Module .WAR file

HTML, GIF, etc.

Servlet JSP

IBM Bindings

IBM Extensions

Web DD

WSDL, Mappings

WebServices DD

IBM WebServices Binding And Extension

WebServices Client DD

IBM WebServices Client Binding & Extension

Client Module .JAR file

Client DD

IBM Binding

WSDL, Mappings

Client Class

WebServices Client DD

IBM WebServices Client Binding & Extension

DD = Deployment Descriptor

Web Services for J2EE: JSR 101 and JSR 109

This is what a web services enabled J2EE application looks like.

There are three potential modules namely the EJB, WAR, and client modules in a J2EE 1.3 application; there are four if you consider the resource adapter.

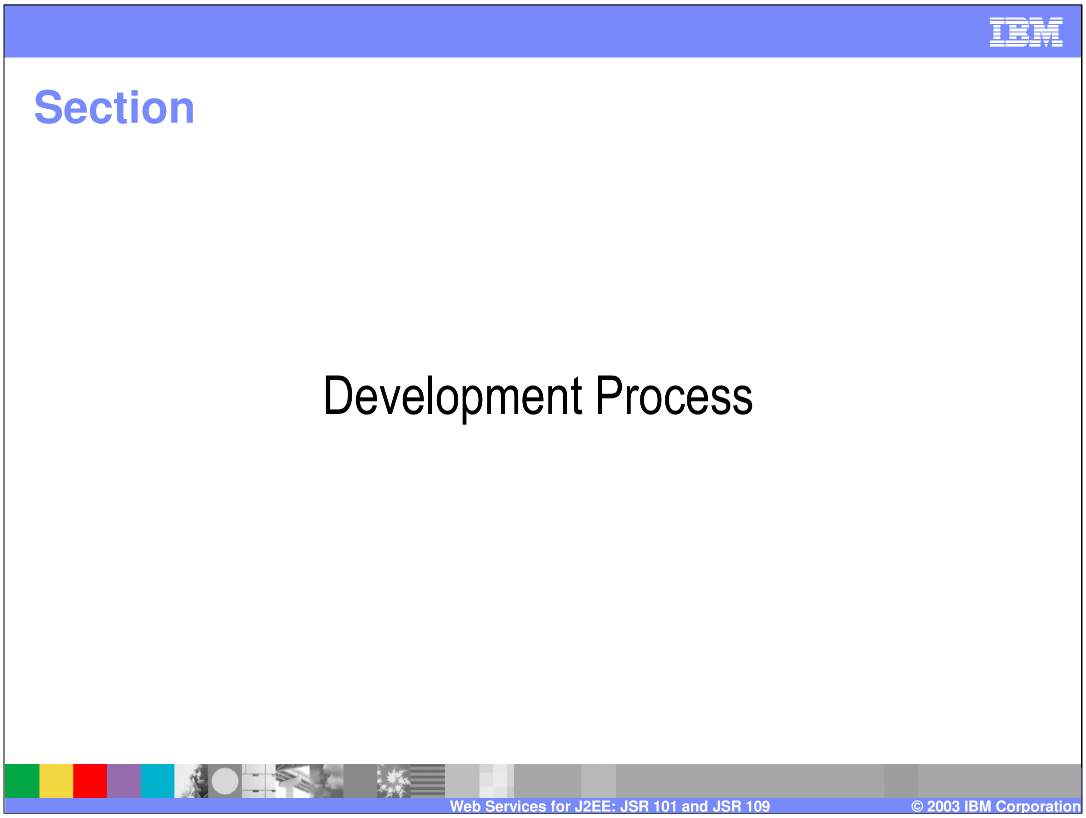
All the non colored white boxes are the standard J2EE 1.3 files and IBM specific files required for non web services information.

What is new as part of web services is all of the colored boxes. Let’s look at the EJB module. If you have exposed an EJB stateless session bean as a web service provider, then the tooling will create a web services deployment descriptor and the IBM extensions and bindings for that web service provider.

Additionally, the WSDL mappings will be created to map the xsd elements to java. If any code within your EJB module is acting as a client to another web service then the webservicesclient.xml deployment descriptor and binding and extensions will need to be created (blue boxes).

The same concept applies to a WAR file. For example, if you have a Java bean that’s bundled with a WAR file then the webservices.xml deployment descriptor along with the binding and extensions will be needed.

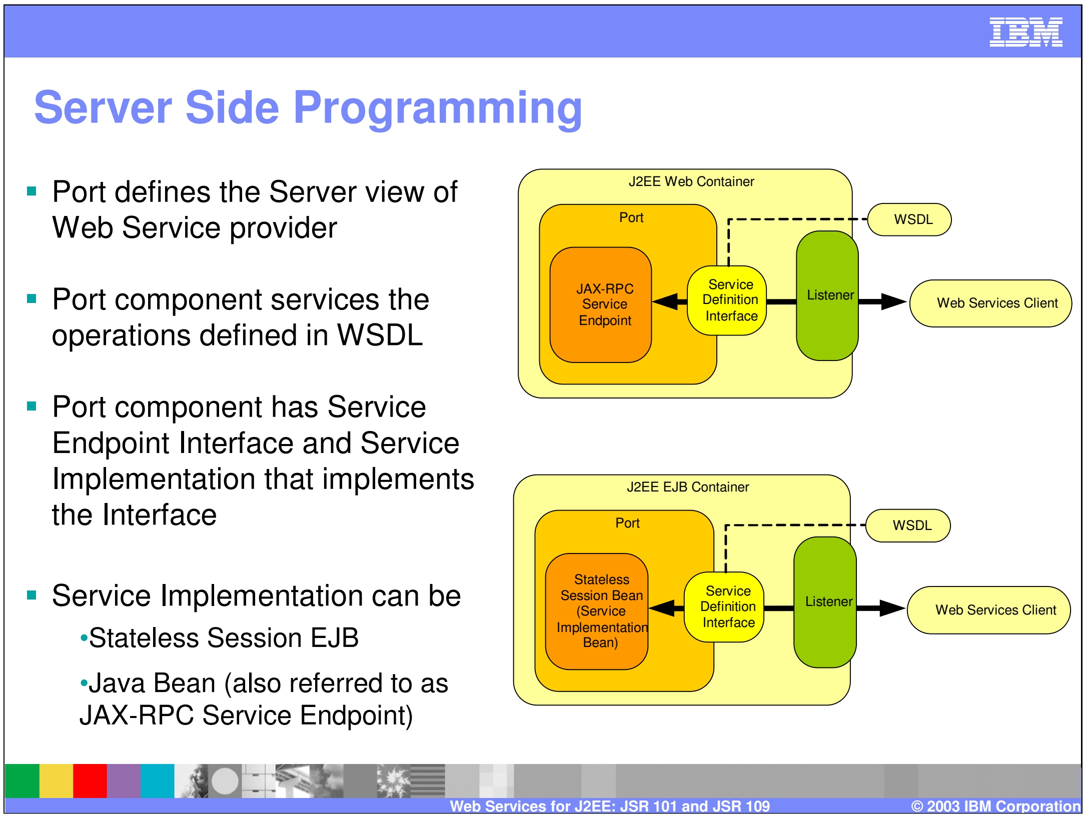
If a servlet or JSP in you web module is acting as a client to another web service, then the associated webservicesclient.xml descriptors and binding and extensions will be needed. In the client module or jar file you will only have the web services client deployment descriptors, bindings and extension along with the WSDL mappings (mapping xsd elements to java and java to xsd elements) because a client module can never act as provider, only as a client. If you want to add/enable WS-Security to your J2EE application then you would have to modify the IBM web services extensions and bindings and the IBM web services client extensions and binding files.



Section

Development Process

Web Services for J2EE: JSR 101 and JSR 109



Server Side Programming

- Port defines the Server view of Web Service provider
- Port component services the operations defined in WSDL
- Port component has Service Endpoint Interface and Service Implementation that implements the Interface
- Service Implementation can be
  - •Stateless Session EJB
  - •Java Bean (also referred to as JAX-RPC Service Endpoint)

Web Services for J2EE: JSR 101 and JSR 109 © 2003 IBM Corporation

Let's talk about Server Side programming.

The first picture shows you a J2EE web container which is used more if you have a Java bean as your service provider. When a Java bean is your service provider this is also called JAX-RPC service endpoint provider. The bottom picture shows you the J2EE EJB container in which you have a stateless session bean as the service provider. The listeners are the components that are different. In the first picture, the listener would be a Web Module - a servlet that gets generated by the Web Services tooling. In the second picture, the listener can be either a servlet for the SOAP/HTTP transport or a message-driven bean that is generated by the endpoint enabler when you select SOAP/JMS as a transport mechanism.

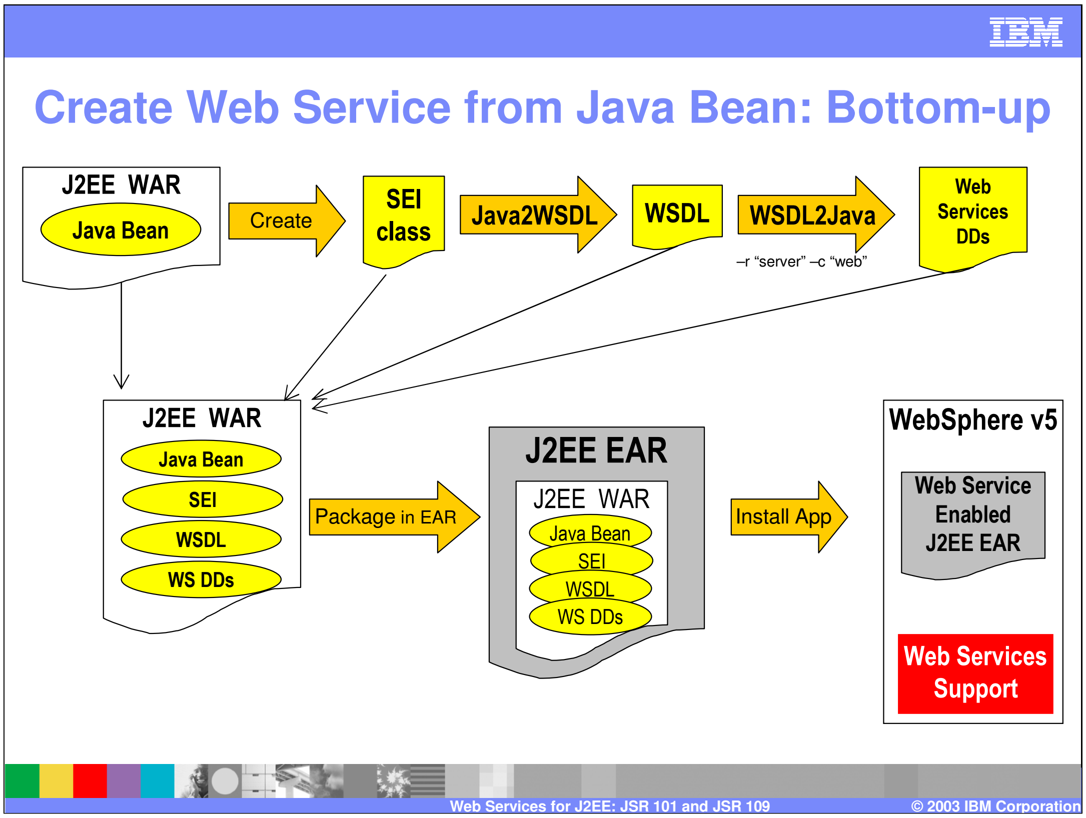
On the server side, you have a port that defines the view of the service provider. This is described by a "service endpoint interface" (SEI) which is the view for the client side. The port component has 2 parts:

1) the service endpoint interface which is the interface of the implementation of the service endpoint which is a Java bean or EJB.
2) the implementation itself which would be the Java bean or EJB.

From a server side perspective, a request is coming in from a listener and the service endpoint interface is what exposes the interface to the client side and through that the web service provider is exposed. You don't have to expose all the methods of the Java bean or EJB. The SEI contains all of the methods that you intend to expose. Typically, the SEI contains a subset of the methods that are already implemented by the implementation artifact (EJB or Java bean).

Service Implementation must have the following requirement

- Class must implement all of the method signatures of the SEI
- Class must be public – not final or abstract
- Must not define finalize() method
- Methods must be public
- For EJB, additional requirements are:
  - Must have default constructor
  - Must implement ejbCreate() and ejbRemove()
  - Must expose the methods via Remote Interface
  - Transaction attributes of the SEI methods must not include Mandatory



Create Web Service from Java Bean: Bottom-up

J2EE WAR

Java Bean

Create

SEI class

Java2WSDL

WSDL

WSDL2Java

–r "server" –c "web"

Web Services DDs

J2EE WAR

Java Bean

SEI

WSDL

WS DDs

Package in EAR

J2EE EAR

J2EE WAR

Java Bean

SEI

WSDL

WS DDs

Install App

WebSphere v5

Web Service Enabled J2EE EAR

Web Services Support

Web Services for J2EE: JSR 101 and JSR 109

Here the process that you need to follow to create a Web Service out of a Java bean is discussed. This process is called "bottom-up creation", because you are starting with the implementation and you do not have the WSDL document to start with. Start in the upper left corner where the Java bean is included in your WAR file. From the Java bean, you create the service endpoint interface Java class. This is a straightforward step. You take the Java bean and remove the methods that you don't want to expose as a web service remembering that you don't have to expose all the methods in the web service. Instead of non-distributed Java exceptions you need to throw Java RemoteException.

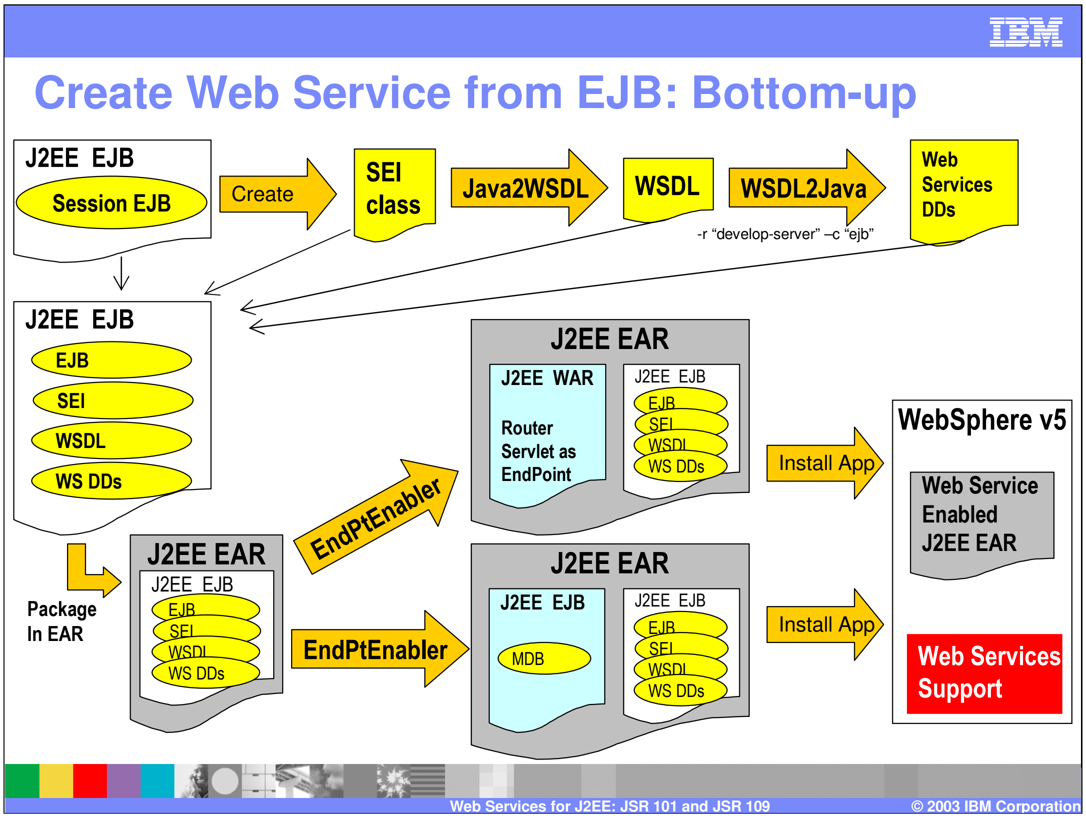
Once the SEI is ready, you would run Java2WSDL on the service endpoint interface and this will create the WSDL document that defines the service endpoint interface. Once you have the WSDL document you will run the WSDL2Java, specifying the role of "server". Another parameter you need to specify is the container type (or –c) which you need to set to "Web" because you would be using the web container and the webservices.xml file that is going to be generated will refer the web.xml file under the covers.

Through the WSDL2Java you will create the web service deployment descriptors and the vendor specific artifacts. As an alternative, you could instead use the –role of "develop-server" and then use WSDeploy at the end – which would result into the same artifacts being generated. In this example, the –r server option is used.

At the end of the generation process, you need to repackage all the files into the J2EE WAR file and then this gets packaged in to the EAR file.

You can then deploy the web services using the WSDeploy utility, or install the EAR file (and run the deployment utility as part of the installation). This web service enabled EAR file application will accept SOAP/HTTP requests.

The final step is the publication of the WSDL document. After installation, the WSDL document carries the exact server name and port that the specific WAR file is listening on. The exported WSDL document can then be used by a client developer to create a service consumer for the Web Service that was just installed.



IBM

Create Web Service from EJB: Bottom-up

J2EE EJB

Session EJB

Create

SEI class

Java2WSDL

WSDL

WSDL2Java

-r "develop-server" –c "ejb"

Web Services DDs

J2EE EJB

EJB

SEI

WSDL

WS DDs

Package In EAR

J2EE EAR

J2EE EJB

EJB

SEI

WSDL

WS DDs

EndPtEnabler

EndPtEnabler

J2EE EAR

J2EE WAR

Router Servlet as EndPoint

J2EE EJB

EJB

SEI

WSDL

WS DDs

J2EE EAR

J2EE EJB

MDB

J2EE EJB

EJB

SEI

WSDL

WS DDs

Install App

Install App

WebSphere v5

Web Service Enabled J2EE EAR

Web Services Support

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

Let's now review the process of making an EJB into a Web Service. You can see the steps are basically the same as for a Java Bean, except that the implementation artifact is a Session EJB. The first difference that is that when you run the WSDL2Java you need to use the –c (or –container) *ejb* value. This will introduce an EJB link into the webservices.xml deployment descriptor, in order to reference the Session EJB that implements the service.

The option to be used for WSDL2Java is "-role develop-server" – the specific WebSphere artifacts (like serializers, deserializers, etc.) will be created later on at deployment time.

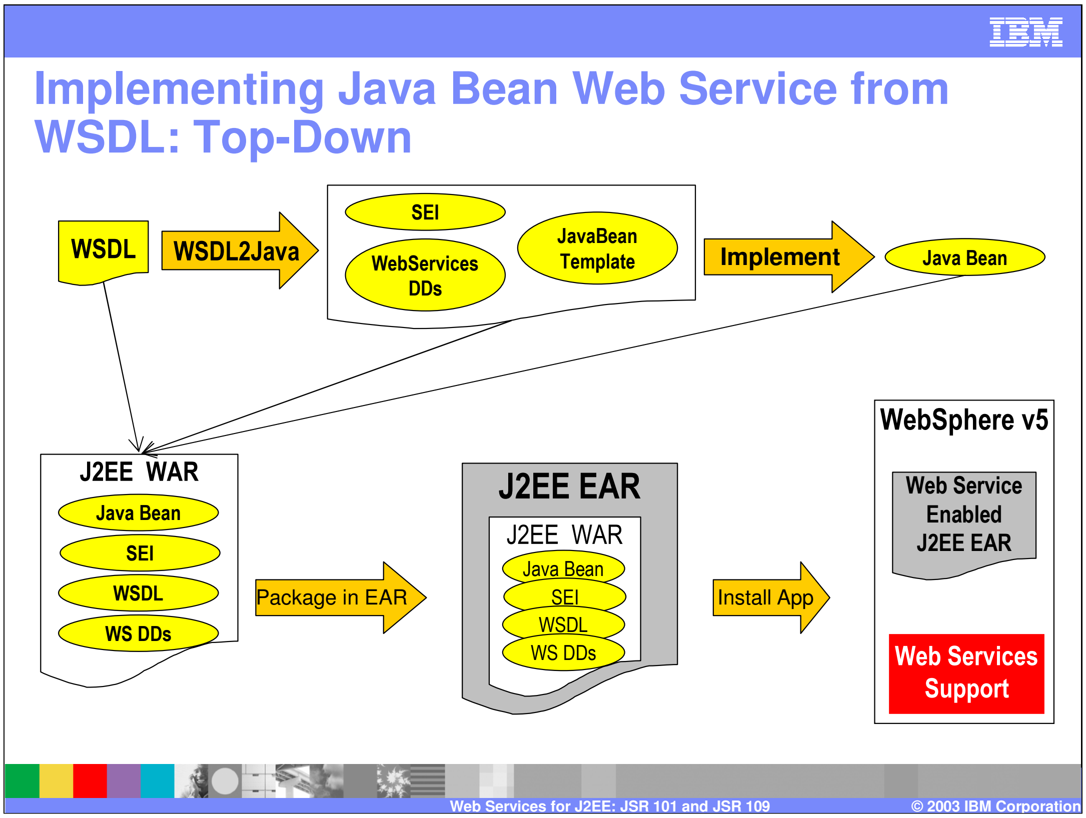
Since the only way an EJB can be called is via RMI/IIOP, you have to create a physical endpoint capable of calling an EJB. That's when the endPtEnabler utility comes into play. This utility can generate the endpoints to be exposed to the clients, and which can call the underlying EJBs.

There are two transport options that you can choose from (the –transport parameter accepts two different values):

1) "–transport http", which is the default. This generates a WAR file that contains the so-called router servlet or web services router servlet. The servlet is the physical endpoint, and will call out to the EJB. This WAR file will be automatically packaged within the EAR file that you are enabling.

2) "–transport jms" is going to create an additional J2EE EJB JAR file that contains the definition of a MDB which is the receiver of the SOAP/JMS request. The MDB will then call the session bean that is located in the original EJB JAR file that contains the service implementation.

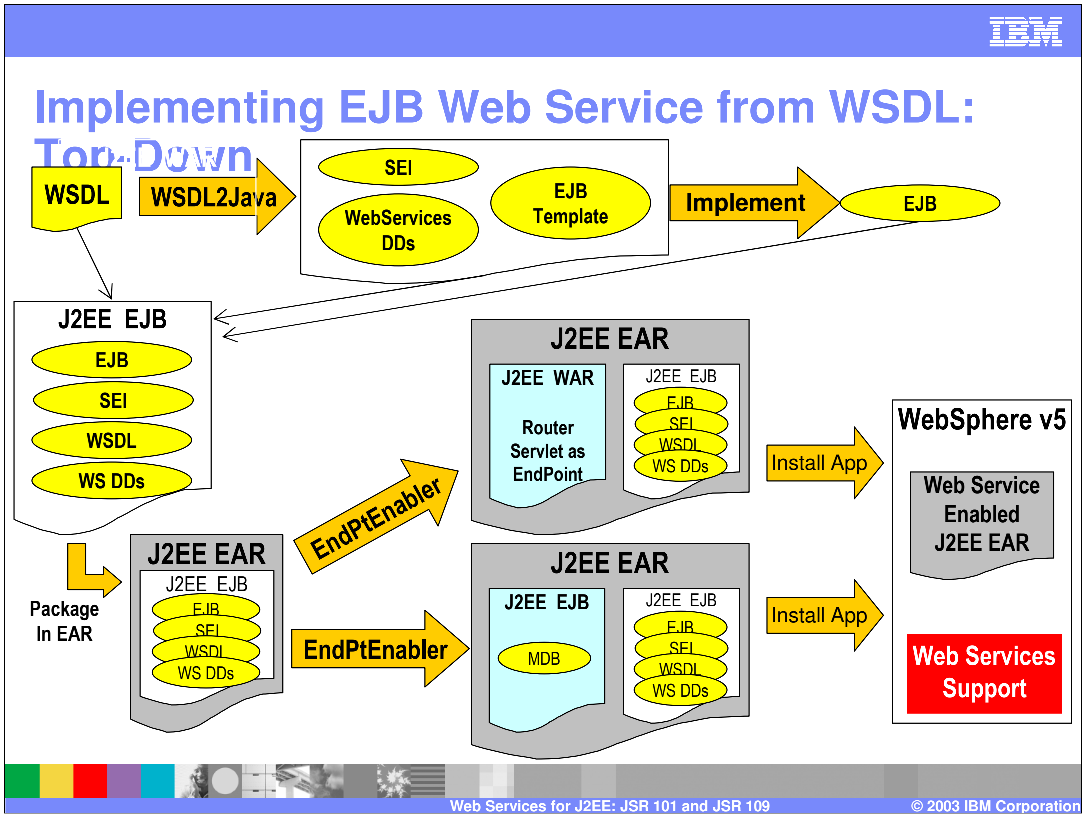
Note that you can use the endpoint enabler to create both module that support SOAP/HTTP and SOAP/JMS at the same time. When you install the EAR file, if you SOAP over JMS is being used, you need to configure the MDB to listen to a specific listener port. That listener port is configured to point to a specific destination (queue or topic) and connection factory. As a developer, you don't need to know which connection factory was going to be used by the administrator so you can now publish the WSDL document which would now contain the exact location of the service provider and you can use this as a starting point for building your client.



Implementing Java Bean Web Service from WSDL: Top-Down

Web Services for J2EE: JSR 101 and JSR 109

Next, a review of building a web service top-down, based on a Java bean implementation, is discussed. You will be starting with a WSDL document that has already been defined, typically as part of a application modeling process. The first tool that you would use would be the WSDL2Java which will create the Java artifacts – the Service Endpoint Interface which corresponds to the operations exposed by the port type that has been defined within the WSDL document itself. In addition, the web services deployment descriptor, and the Java bean template will be generated. With the Java bean implementation ready all you will have to do is to fill in the business logic for the specific methods that have been exposed as a web service provider. You will then package all the pieces into a WAR file and ultimately into an EAR file. The final step is to install the EAR file into the WebSphere Application Server.



Implementing EJB Web Service from WSDL: Top-Down

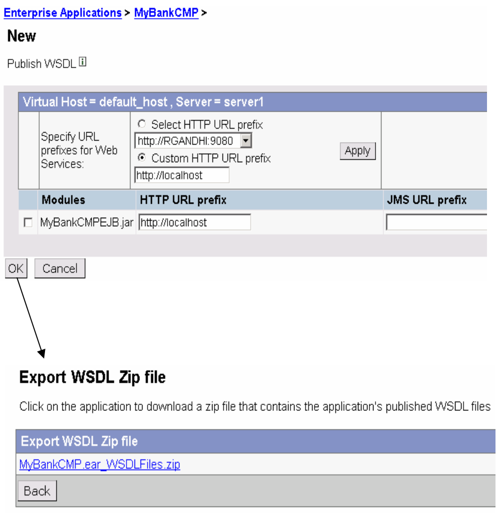
Web Services for J2EE: JSR 101 and JSR 109

The last high-level process discussed is the top-down creation of a Web Service implemented by an EJB. Notice that the WSDL2Java with the "–container ejb" option is used because an EJB has to be created from the WSDL document. This operation will create the service endpoint interface which will act as the interface of your EJB implementation. It will also create the webservices.xml and IBM-specific bindings and extensions files. It will also generate the home and remote interface and a template for the EJB class. You will still have to implement the EJB logic for the those methods which were exposed in the WSDL document as operations. Finally, the artifacts are packaged into an EAR file and installed into the runtime.

Publishing WSDL

- After installing Web Service enabled EAR, use the “Publish WSDL” System Management feature to publish a well defined WSDL that includes the service location
  - Using Admin Console or wsadmin
- Used to create Web Service client

Application -> Enterprise Application -> Publish WSDL



Enterprise Applications > MyBankCMP >

New

Publish WSDL

Virtual Host = default_host , Server = server1

Specify URL prefixes for Web Services:	○ Select HTTP URL prefix http://RGANDHI:9080 ● Custom HTTP URL prefix http://localhost	Apply

	Modules	HTTP URL prefix	JMS URL prefix
□	MyBankCMPEJB.jar	http://localhost	

Export WSDL Zip file

Click on the application to download a zip file that contains the application's published WSDL files

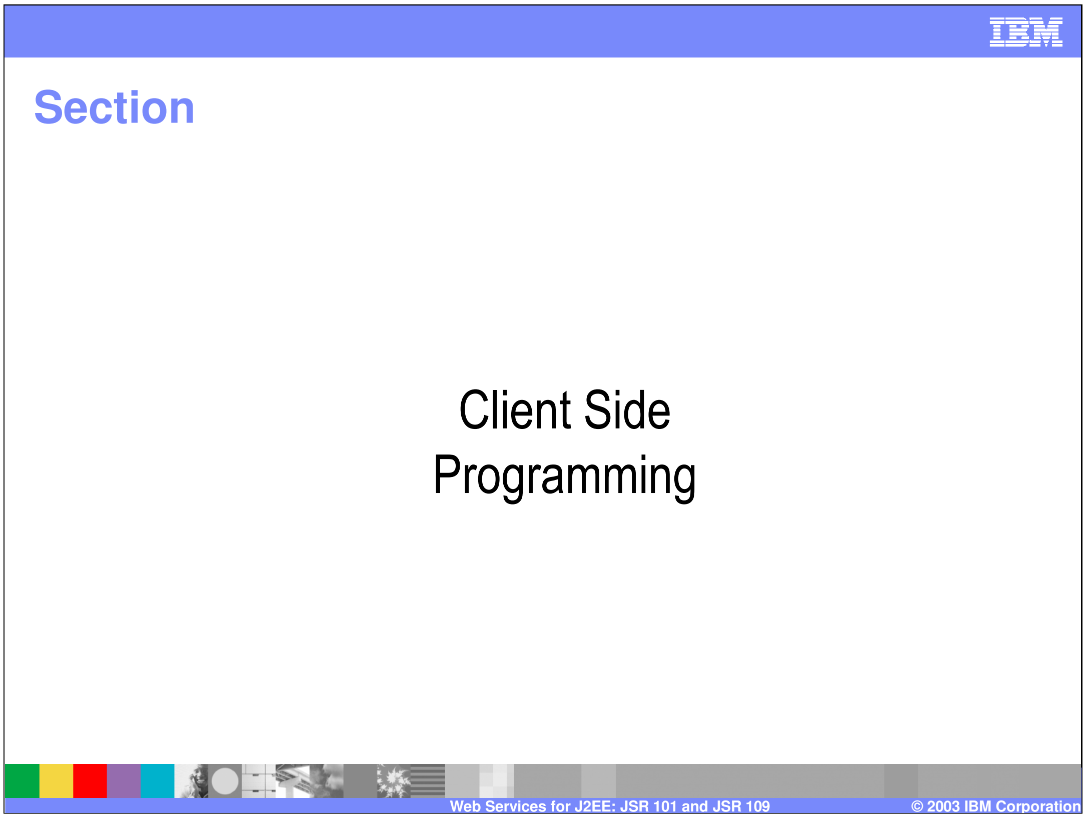
Export WSDL Zip file
MyBankCMP.ear_WSDLFiles.zip

Once you have installed the web service enabled EAR file into the application server, one or more module of the EAR file have a webservices.xml indicating that there is piece of code that will act as a service provider.

Normally, when developers creates a WSDL document they do not need to know about the end location of where the service will be physically hosted (HTTP ports, host name, queue connection factory and destinations, listener ports for MDBs….)

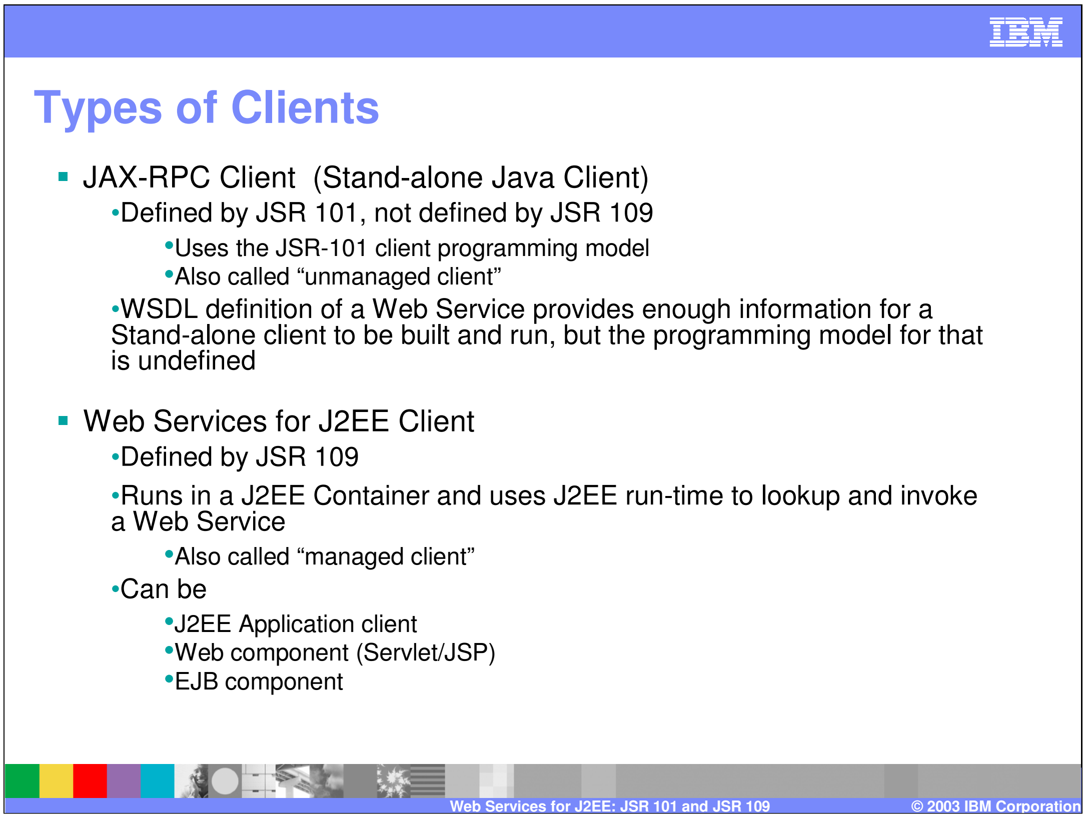
Once the system administrator installs the application, all that info is fully defined. That’s why there is an additional administrative step that allows the administrator to export (or publish) the WSDL document with all that information, so that a service client can utilize it to create its proxy to the service.

This chart shows you the detail of the console in the act of exporting the WSDL file.



Section

Client Side Programming



Types of Clients

- JAX-RPC Client (Stand-alone Java Client)
  - Defined by JSR 101, not defined by JSR 109
    - Uses the JSR-101 client programming model
    - Also called “unmanaged client”
  - WSDL definition of a Web Service provides enough information for a Stand-alone client to be built and run, but the programming model for that is undefined
- Web Services for J2EE Client
  - Defined by JSR 109
  - Runs in a J2EE Container and uses J2EE run-time to lookup and invoke a Web Service
    - Also called “managed client”
  - Can be
    - J2EE Application client
    - Web component (Servlet/JSP)
    - EJB component

Web Services for J2EE: JSR 101 and JSR 109

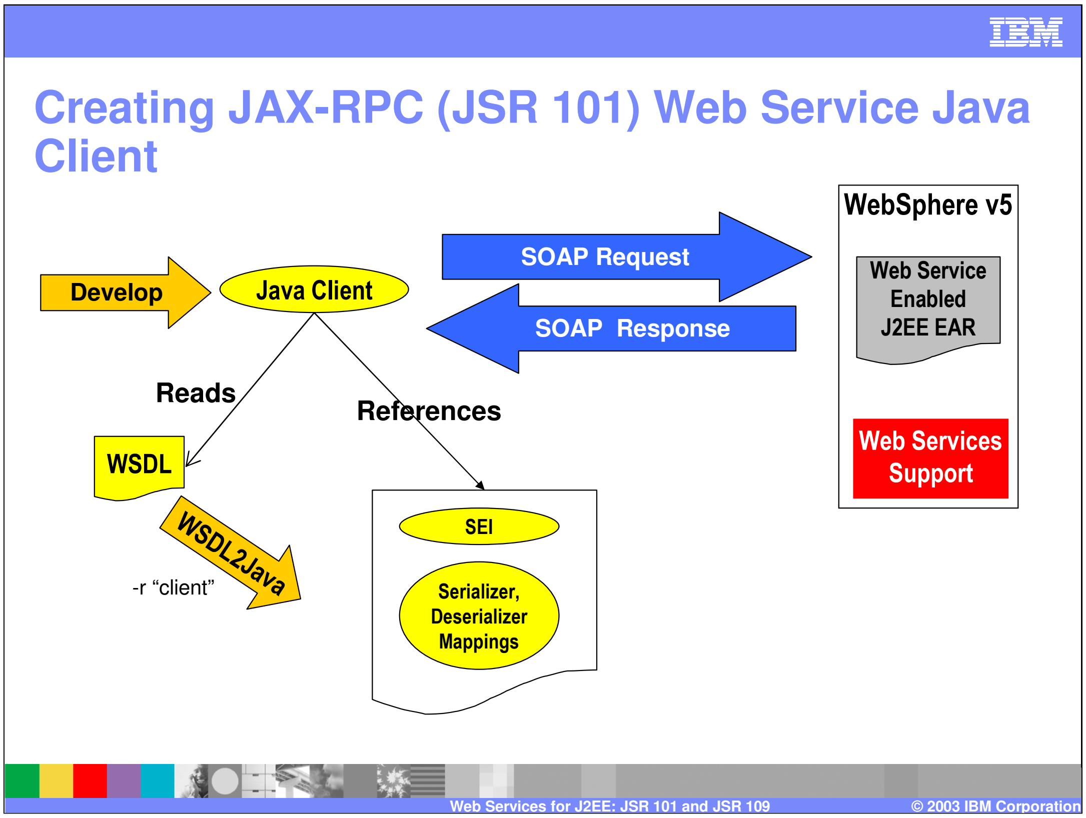
© 2003 IBM Corporation

When discussing “Client-side programming” in JSR 101/109, this implies specifically Java Web Services clients.

There are fundamentally two types of clients:

1) Stand-alone Java client which is also referred to as a JAX-RPC client. These clients directly inspect a WSDL file and formulate the calls to the Web Service by using the JAX-RPC APIs directly. These clients are packaged as plain Java JAR files, which do not contain any deployment information. These clients do not run in any J2EE container. Since these clients run outside of a J2EE container, they are also called “unmanaged clients”. In order to invoke these clients, you would have to ensure that the JAR file is in the CLASSPATH and then use the *java* command to invoke the program.

2) JSR 109 clients run inside a J2EE container. These clients are packaged as EAR files and contain components that act as service requesters. These components maybe J2EE client applications, in which case you would use the *launchClient* tool to invoke them. Or, they may be server side components (servlets, Session EJBs) which need to call out to a Web Service. In both cases, these clients would use the JSR 109 APIs and deployment information to lookup and invoke the service.



Creating JAX-RPC (JSR 101) Web Service Java Client

Web Services for J2EE: JSR 101 and JSR 109

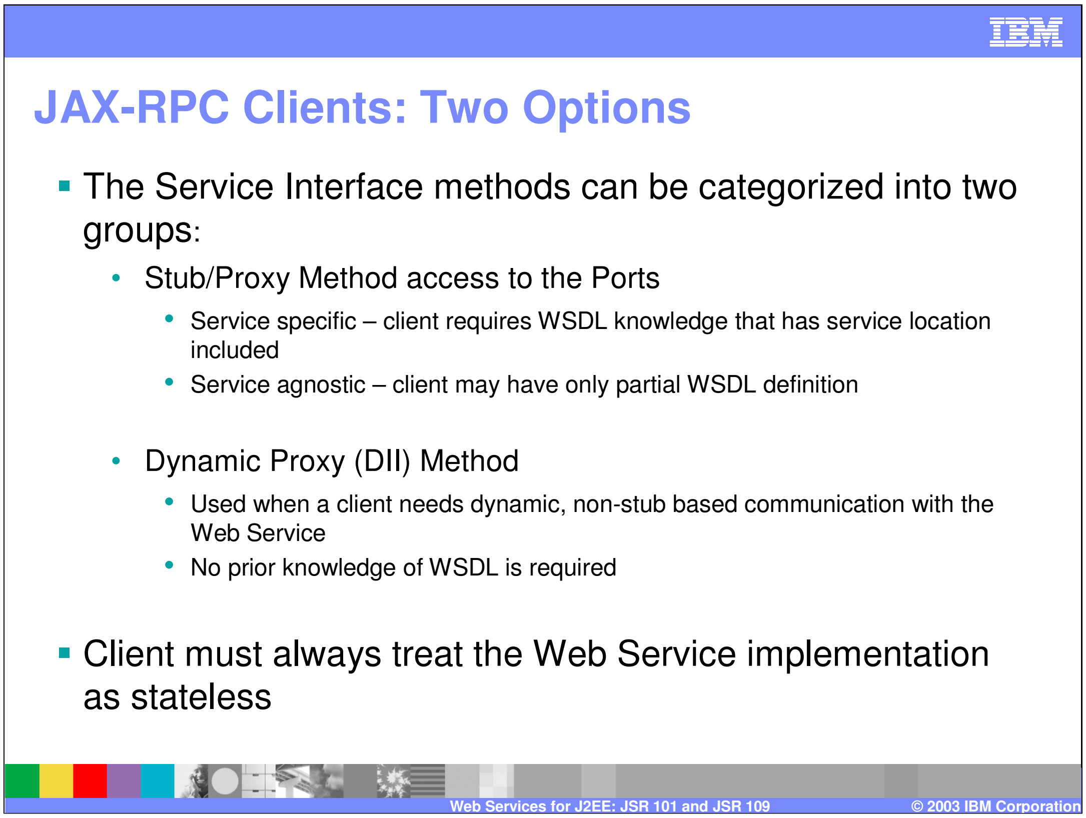
This chart discusses how to create a JAX-RPC Client.

You can start with a WSDL document which may have been given to you by the administrator – or discovered out of a UDDI registry.

You then run the WSDL2Java with the –r client option which will create the service endpoint interface. It also generates the serializer, the deserializer, and the data type mappings artifacts. You can now package all these artifacts together in a JAR file and run the client to exchange SOAP messages with the service provider.

WSDL2Java options:

WSDL2java -r “client” <wsdl file> → this will create the binding and the development files used for Unmanaged clients



JAX-RPC Clients: Two Options

- The Service Interface methods can be categorized into two groups:
  - Stub/Proxy Method access to the Ports
    - Service specific – client requires WSDL knowledge that has service location included
    - Service agnostic – client may have only partial WSDL definition
  - Dynamic Proxy (DII) Method
    - Used when a client needs dynamic, non-stub based communication with the Web Service
    - No prior knowledge of WSDL is required
- Client must always treat the Web Service implementation as stateless

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

<u>Stub/Proxy Access:</u>

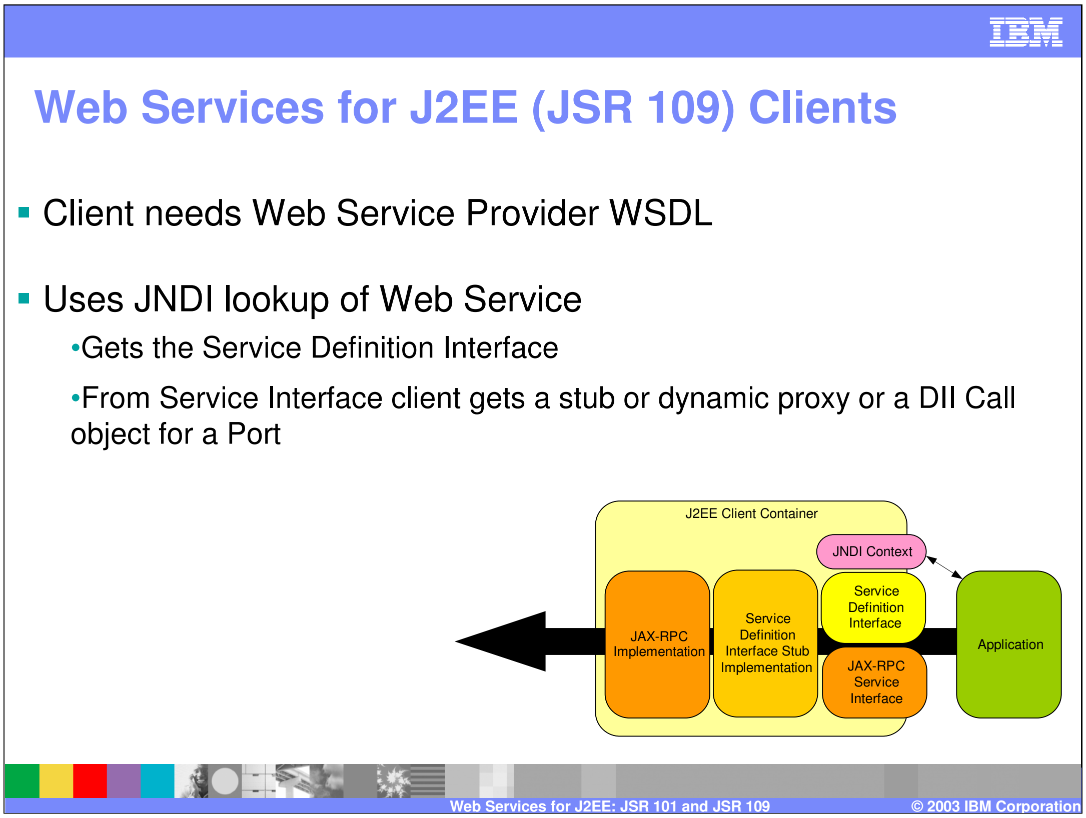
The client may use the following Service Interface methods to obtain a static stub or dynamic proxy for a Web service:

```
java.rmi.RemotegetPort(QNameportName,ClassserviceEndpointInterface)throwsServiceException;

java.rmi.RemotegetPort(java.lang.ClassserviceEndpointInterface)throwsServiceException;
```

<u>Dynamic Port Access</u>

A client may use the following DII methods of a Service Interface located by a JNDI lookup of the client's environment to obtain a Call object:

```
Call createCall()throwsServiceException;

Call createCall(QNameportName)throwsServiceException;

Call createCall(QNameportName,StringoperationName)throwsServiceException;

Call createCall(QNameportName,QNameoperationName)throwsServiceException;

Call[] getCalls(QNameportName)throwsServiceException;
```



Web Services for J2EE (JSR 109) Clients

- Client needs Web Service Provider WSDL
- Uses JNDI lookup of Web Service
  - Gets the Service Definition Interface
  - From Service Interface client gets a stub or dynamic proxy or a DII Call object for a Port

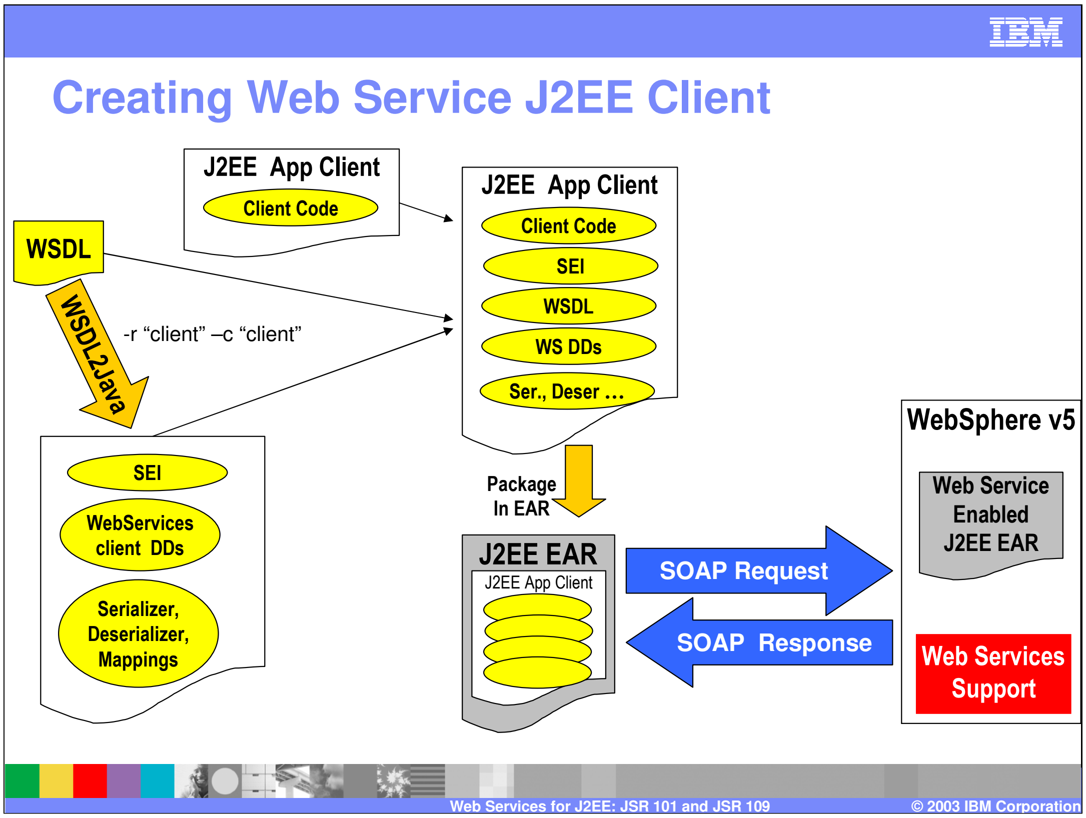
J2EE Client Container
JNDI Context
JAX-RPC Implementation
Service Definition Interface Stub Implementation
Service Definition Interface
JAX-RPC Service Interface
Application

Web Services for J2EE: JSR 101 and JSR 109

In JSR 109, the client always uses the service provider's WSDL file as a reference for invoking a service. Then, based on the information stored in the webservicesclient.xml and in the bindings file, it will look up a service using JNDI. It will get back a Service Definition Interface, that can be used to transparently invoke the service, either via a stub, or by using the dynamic invocation (more seldom).

A reference to the Web Service implementation should never be passed to another object. A client should never access the Web Service implementation directly. Doing so bypasses the container's request processing which may open security holes or cause anomalous behavior.

Client cannot distinguish whether the methods are being performed locally or remotely, nor can the client distinguish how the service is implemented.

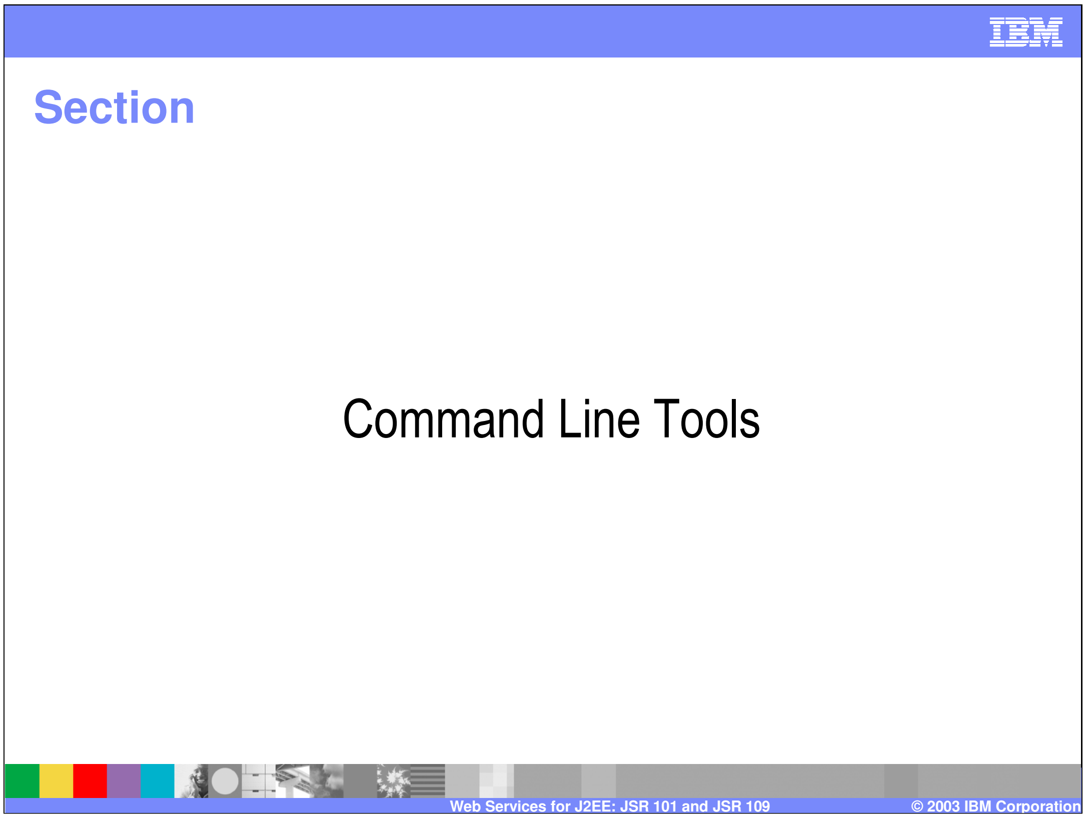


These are the basics steps needed to generate a JSR 109 Client:

1) Based on the WSDL file published by the service provider, use the WSDL2Java tool to create the Service Endpoint Interface, the client deployment descriptor, and any serializer/deserializer and mappings that may be necessary

2) The client code has to be developed separately and packaged with the artifacts you just generated. In our example, a simple J2EE client is shown, but conceivably, this process applies to other types of Java Web Services clients (WARs, EJBs).

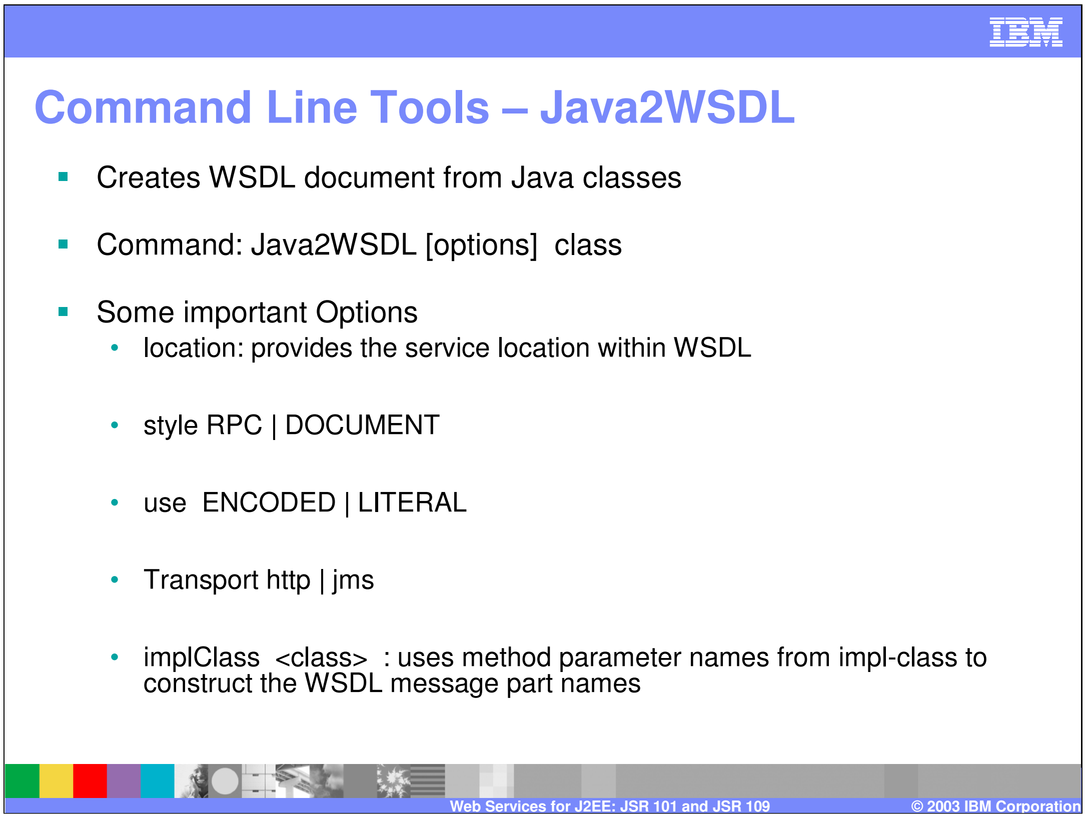
3) Once the packaging is complete, you can use launchClient to invoke the client application.



Section

Command Line Tools

Web Services for J2EE: JSR 101 and JSR 109



Command Line Tools – Java2WSDL

- Creates WSDL document from Java classes
- Command: Java2WSDL [options] class
- Some important Options
  - location: provides the service location within WSDL
  - style RPC | DOCUMENT
  - use ENCODED | LITERAL
  - Transport http | jms
  - implClass <class> : uses method parameter names from impl-class to construct the WSDL message part names

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

There are four command line tools that can help you in creating JSR 101/109 applications. The first one discussed is

Java2WSDL. It generates a WSDL file based on an existing SEI class. This command is useful in the bottom-up process.

One important parameter is the service location: the WSDL document needs to have the service location because the client will need to know which server the service is located on, which port it is listening on, or which JMS destination to send the message to.

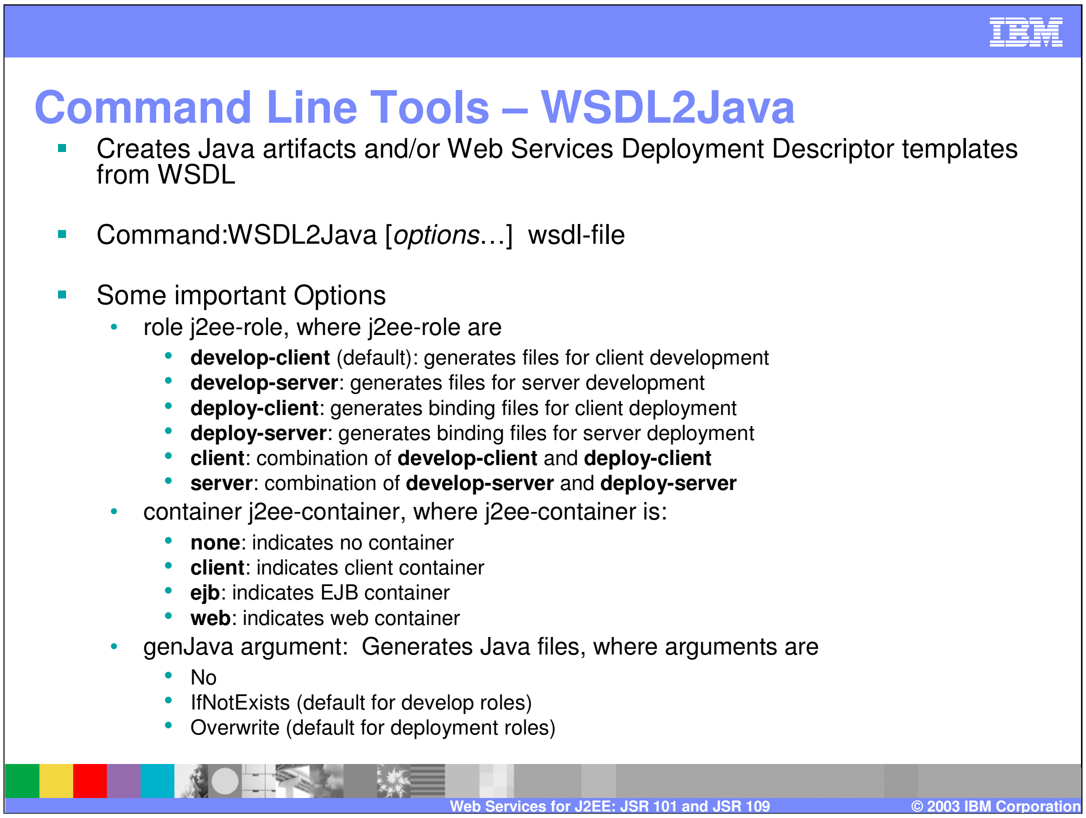
A developer may not know the service location at the time the command is executed. In that case, a “dummy” location should be specified. During the deployment process, you will then be able to fully specify the location.

The style parameter allows you to choose between RPC or doc literal.

Doc literal is becoming more predominant, but rpc, rpc literal and rpc encoded are still supported.

Another important option is what type of transport do you want to use such as HTTP, JMS or both. You can create a WSDL document that only uses SOAP over HTTP or SOAP over JMS.

Finally, the implementation class points to the actual Java bean or EJB bean class that implements the service.



Command Line Tools – WSDL2Java

- Creates Java artifacts and/or Web Services Deployment Descriptor templates from WSDL
- Command:WSDL2Java [*options*…] wsdl-file
- Some important Options
  - role j2ee-role, where j2ee-role are
    - **develop-client** (default): generates files for client development
    - **develop-server**: generates files for server development
    - **deploy-client**: generates binding files for client deployment
    - **deploy-server**: generates binding files for server deployment
    - **client**: combination of **develop-client** and **deploy-client**
    - **server**: combination of **develop-server** and **deploy-server**
  - container j2ee-container, where j2ee-container is:
    - **none**: indicates no container
    - **client**: indicates client container
    - **ejb**: indicates EJB container
    - **web**: indicates web container
  - genJava argument: Generates Java files, where arguments are
    - No
    - IfNotExists (default for develop roles)
    - Overwrite (default for deployment roles)

Web Services for J2EE: JSR 101 and JSR 109

The second tool is **WSDL2Java.** If you have a WSDL document, WSDL2Java creates the java artifacts and the deployment descriptors webservices.xml and webservices-client.xml that need to be included within your Java web services modules.

In a bottom up approach, you use the Java2WSDL to create a WSDL document and then you run the WSDL2Java and create the webservices.xml and IBM specific files. The different roles that you can choose determine whether the webservices.xml or webservices-client.xml file are created.

The tool also creates WebSphere specific files like the stubs and serializers and deserializers.

WSDL2Java can be used to create both the client and the server artifacts. For example, if you select the role as server if create the webservices.xml and the vendor specific server side files.

The container option determines what kind of container you are trying to create the Java artifacts for – e.g., client, EJB container, or web container.

If you are creating Web Services out of Java beans, then the Java bean has to be included in a WAR file – the webservices.xml will have the appropriate tags to inspect the web.xml file. If you're creating a service out of an EJB stateless session bean then the EJB container will be used and the webservices.xml will have appropriate links to the EJB that will be exposed as a web service.

Last if you are creating for a client container then there will be appropriate links to examine the applicationclient.xml.

Another option is the genJava argument. If you select *no,* you will only create deployment descriptors and not create the vendor specific files.

WSDL2Java generates the following classes and files:

For each portType in the WSDL document (<WSDL:portType> element tag):

•A Service Endpoint Interface is generated.

For each service in the WSDL document (<wsdl:service> element tag):

•a Service Interface is generated.

•A ServiceLocator is generated. This class is a WebSphere-specific implementation of the Service interface, and is not used directly by developers.

•A webservices.xml deployment descriptor template for the endpoint implementation.

•An ibm-webservices-bnd.xmi deployment descriptor template.

•A *wsdlfile*_mapping.xml deployment descriptor template.

•A META-INF/webservicesclient.xml deployment descriptor template for use by a client.

•A META-INF/ibm-webservicesclient-bnd.xmi deployment descriptor template

•A META-INF/webservices.xml deployment descriptor template for use by a client.

•A META-INF/ibm-webservices-bnd.xmi deployment descriptor template

For each binding in the WSDL document (<wsdl:binding> element tag):

•A stub is generated that implements the Service Endpoint Interface.

•When the -role server and -container Ejb options are specified an implementation template for an enterprise bean and templates for the EJB remote interface and home are generated.

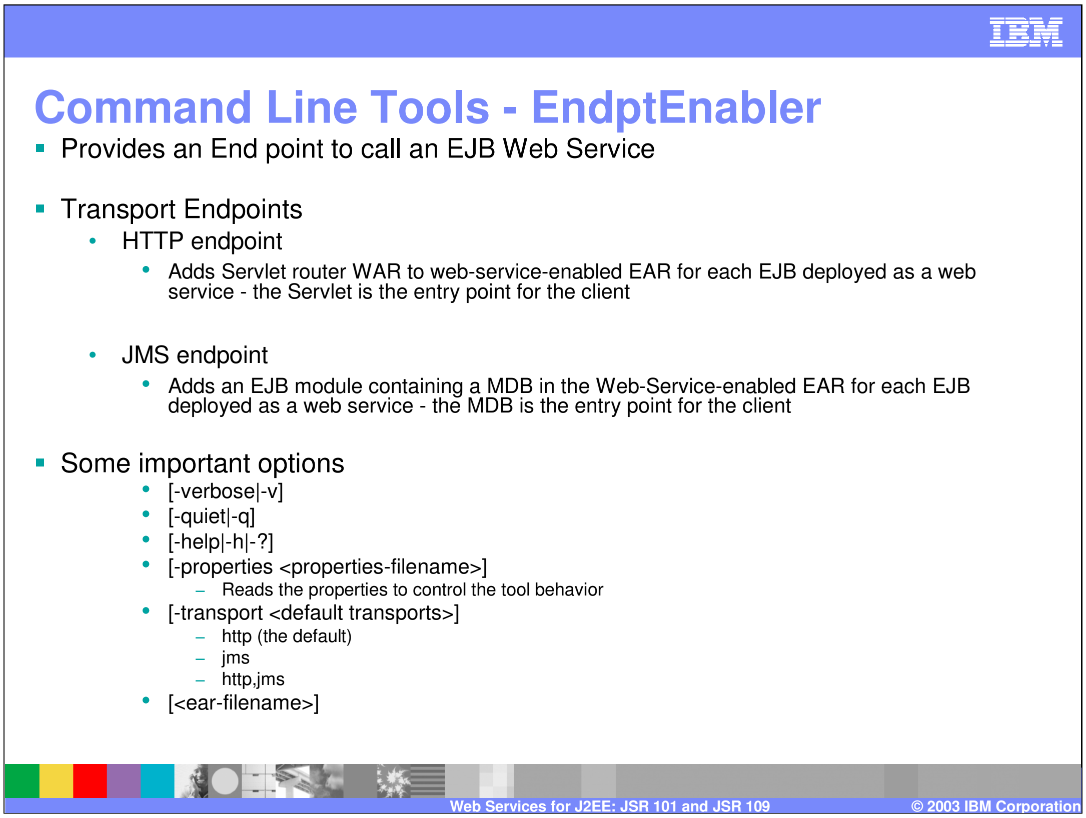
•When the -role server and -container Web options are specified an implementation template for the Java bean is generated.

Also generated:

•For each complexType or simpleType, a Java Bean representing the structure of the type.

•For each complexType, 3 classes (*_Ser.java, *_Deser.java, and *_Helper.java) to assist in converting the bean to SOAP and back.

•For each out/inout parameter, a Holder class



Command Line Tools - EndptEnabler

- Provides an End point to call an EJB Web Service
- Transport Endpoints
  - HTTP endpoint
    - Adds Servlet router WAR to web-service-enabled EAR for each EJB deployed as a web service - the Servlet is the entry point for the client
  - JMS endpoint
    - Adds an EJB module containing a MDB in the Web-Service-enabled EAR for each EJB deployed as a web service - the MDB is the entry point for the client
- Some important options
    - [-verbose|-v]
    - [-quiet|-q]
    - [-help|-h|-?]
    - [-properties <properties-filename>]
      - Reads the properties to control the tool behavior
    - [-transport <default transports>]
      - http (the default)
      - jms
      - http,jms
    - [<ear-filename>]

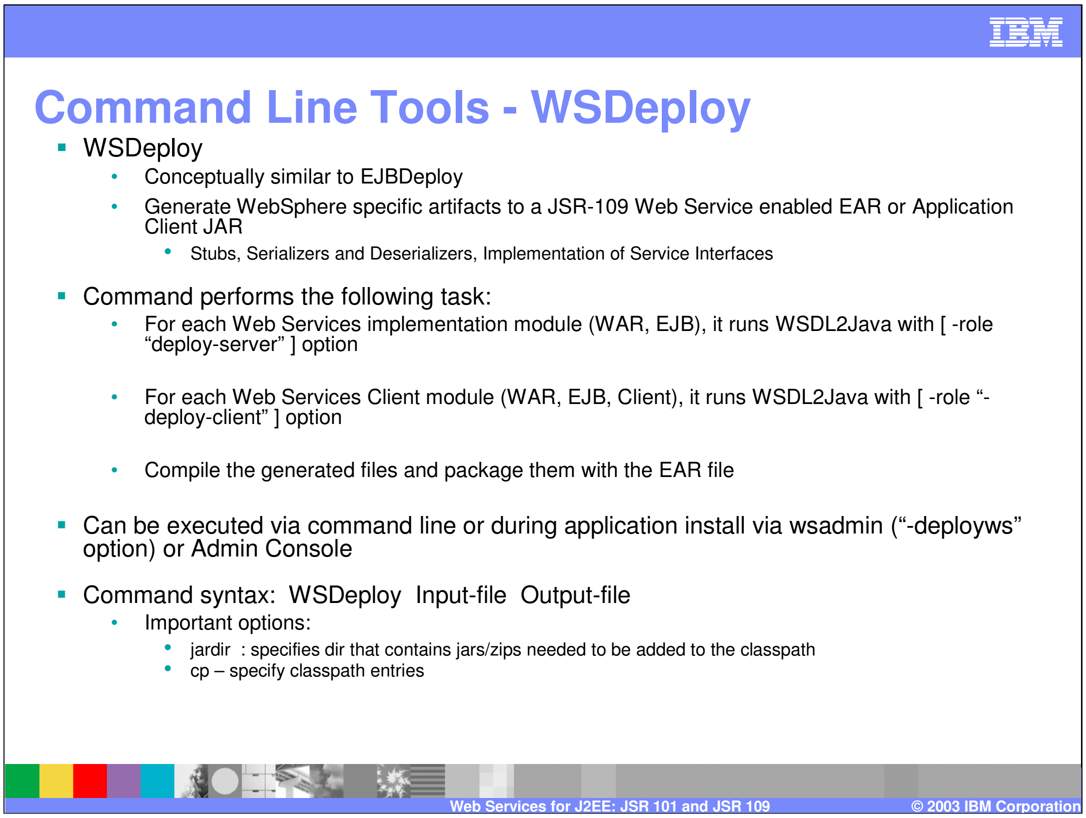
Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

EndPtEnabler only applies to Web Services based off a Session EJB. As previously articulated, EJBs can only be called via RMI/IIOP, so EndPtEnabler creates a Web module or a Message-driven Bean that will take care of invoking the EJB on behalf of the Web Services client.

Both HTTP and JMS endpoints are therefore possible. The default is HTTP, which assumes that you will be calling that web service EJB through SOAP/HTTP. In that case, a WAR module will be added to the EAR file you are trying to enable.

If you specify JMS for the -transport option, the assumption is that you will be calling the EJB using SOAP over JMS and create another endpoint which will be a Message-Driven bean. Or you can choose both the HTTP endpoint and MDB JMS endpoint receiver.



Command Line Tools - WSDeploy

- WSDeploy
  - Conceptually similar to EJBDeploy
  - Generate WebSphere specific artifacts to a JSR-109 Web Service enabled EAR or Application Client JAR
    - Stubs, Serializers and Deserializers, Implementation of Service Interfaces
- Command performs the following task:
  - For each Web Services implementation module (WAR, EJB), it runs WSDL2Java with [-role "deploy-server"] option
  - For each Web Services Client module (WAR, EJB, Client), it runs WSDL2Java with [-role "-deploy-client"] option
  - Compile the generated files and package them with the EAR file
- Can be executed via command line or during application install via wsadmin ("-deployws" option) or Admin Console
- Command syntax: WSDeploy Input-file Output-file
  - Important options:
    - jardir : specifies dir that contains jars/zips needed to be added to the classpath
    - cp – specify classpath entries

Web Services for J2EE: JSR 101 and JSR 109

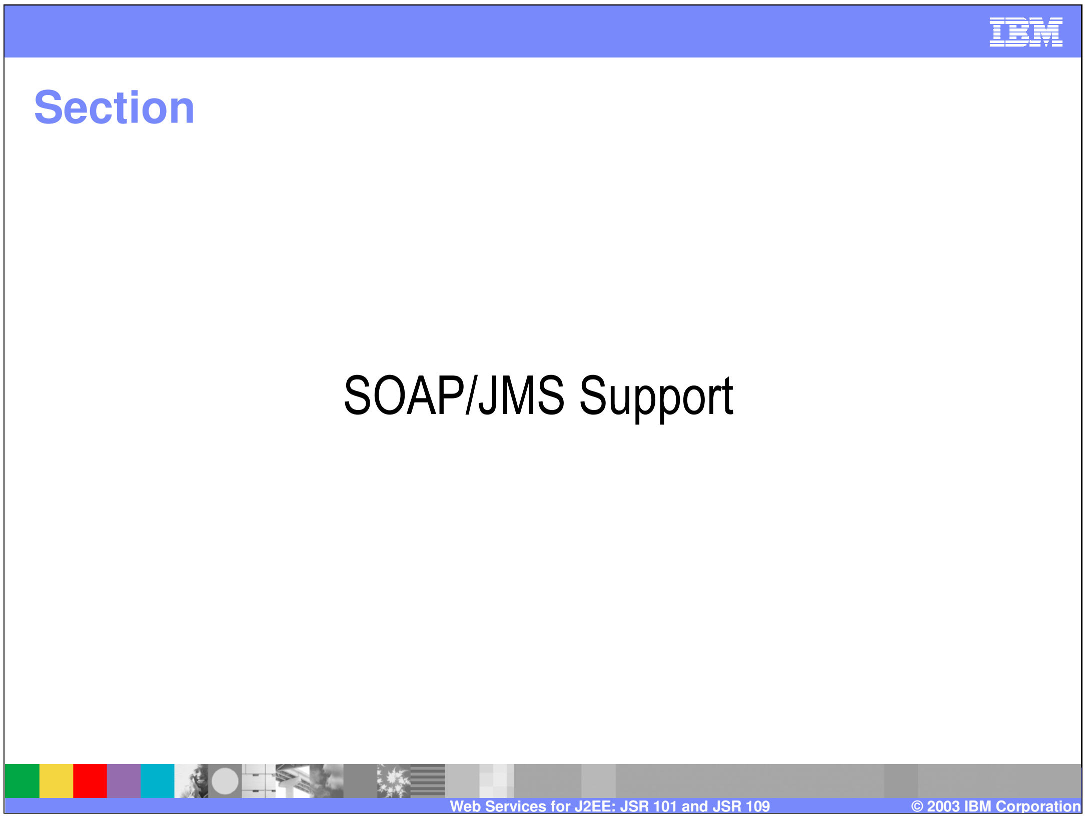
© 2003 IBM Corporation

The last command line tool is the WSDeploy tool. WSDeploy creates WebSphere specific artifacts for the Web services enabled EAR file.

This tool is useful if you did not already selected the deployment options when you ran WSDL2Java.

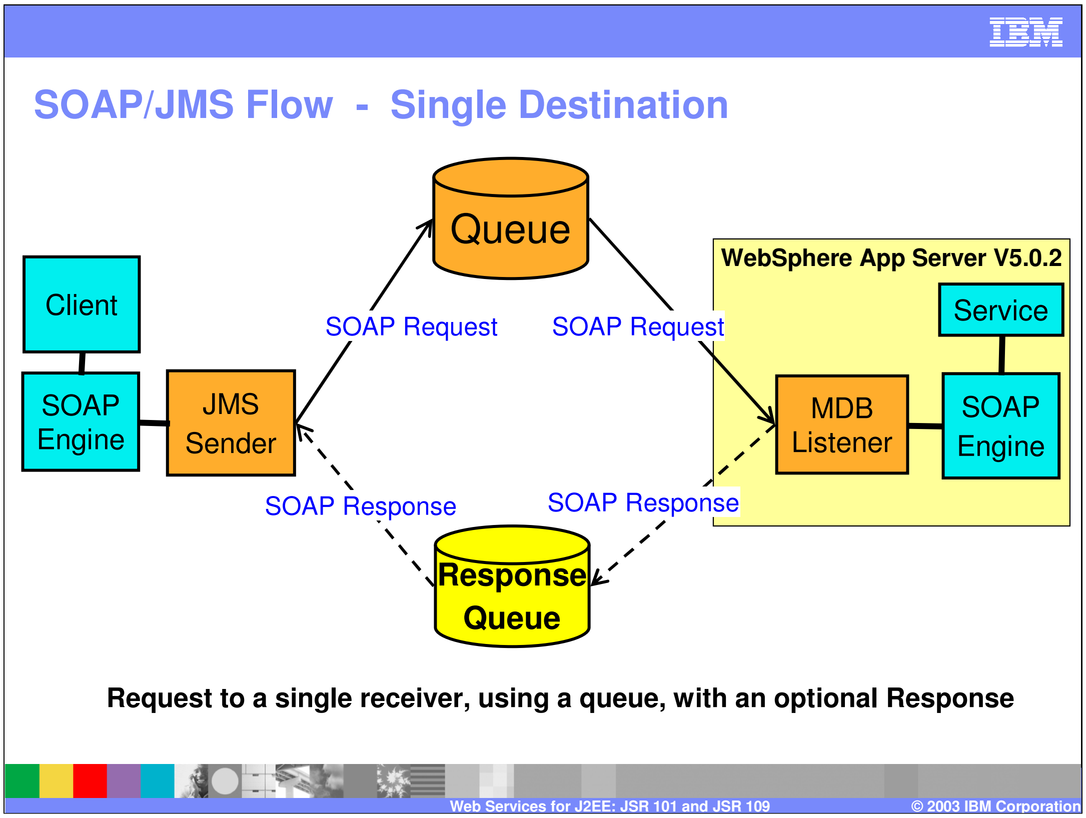
WSDeploy will run the WSDL2Java command for each web services implementation modules (the WAR file and EJB file for the provider) using the role of deploy server which will create the vendor specific stubs and serializers deserializer and the implementation of the services interfaces. For each of the Web Services Client modules (web module or EJB modules acting as clients) what it looks for is if that module has a webservices-client.xml file, it will run the WSDL2Java command with the –deploy-client option.

You can also run WSDeploy right at the time the application is installed on the Application Server. A specific installation option allows you to do that.



Section

SOAP/JMS Support



SOAP/JMS Flow - Single Destination

Request to a single receiver, using a queue, with an optional Response

Web Services for J2EE: JSR 101 and JSR 109

© 2003 IBM Corporation

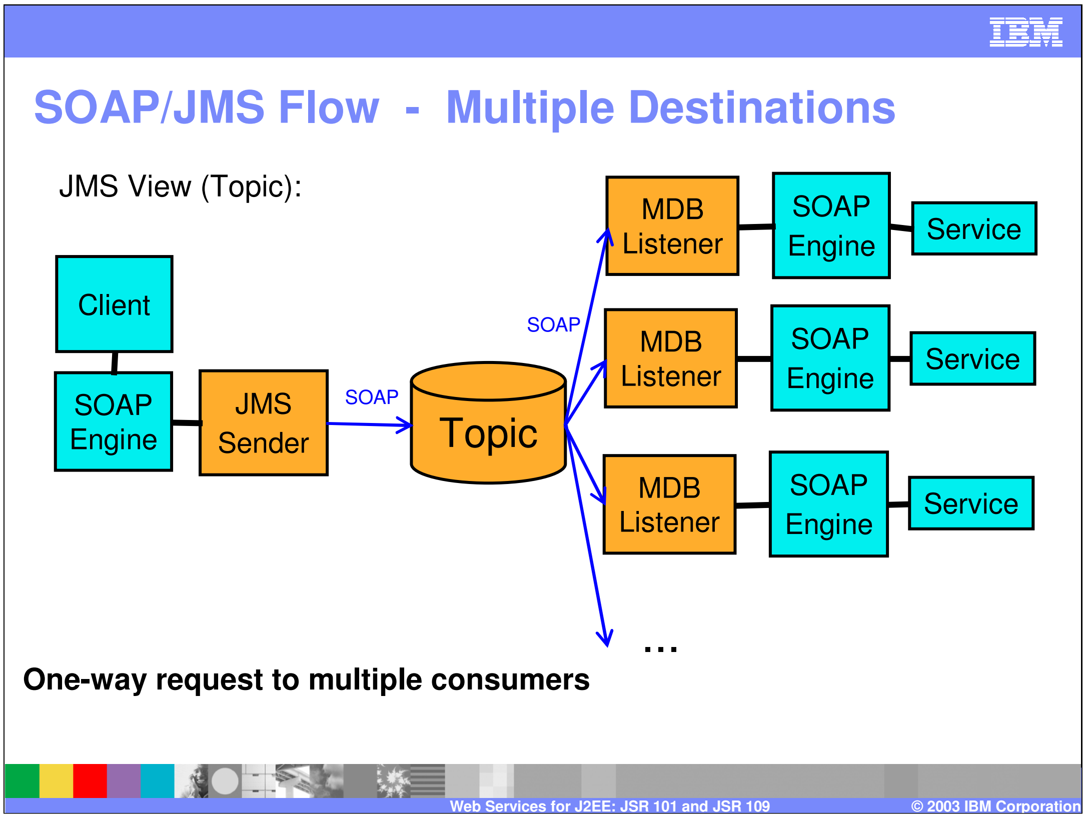
WebSphere Application Server V5.0.2 allows SOAP messages to flow over the JMS transport layer. At this stage of SOAP/JMS support, there are two options that are available:

1) You expose a one way operation using SOAP/JMS (no response required). The service requester will invoke the service by sending a SOAP message to a JMS destination (as shown in the picture).

2) You expose a two ways operation with synchronous response. The service requester sends a message to the JMS destination (a queue), just like in the previous case. But then, it sits on a temporary response queue waiting for the response to be produced. The fully asynchronous model is not yet supported by V5.0.2.

In either case, you just need to set up the JMS destination to store the inbound request messages.

On the server side, the request is going to be handled by a Message-Driven bean, which acts as the SOAP engine for SOAP/JMS. The MDB is going to be generated by the end point enabler tool (endPtEnabler). On the client side, the WSDL2Java tool will generate a J2EE client that uses the JMS APIs to interact with the messaging resources (JMS is available to the client container). If a response is required, the Message-Driven bean is going to produce it and send it the temporary response queue.



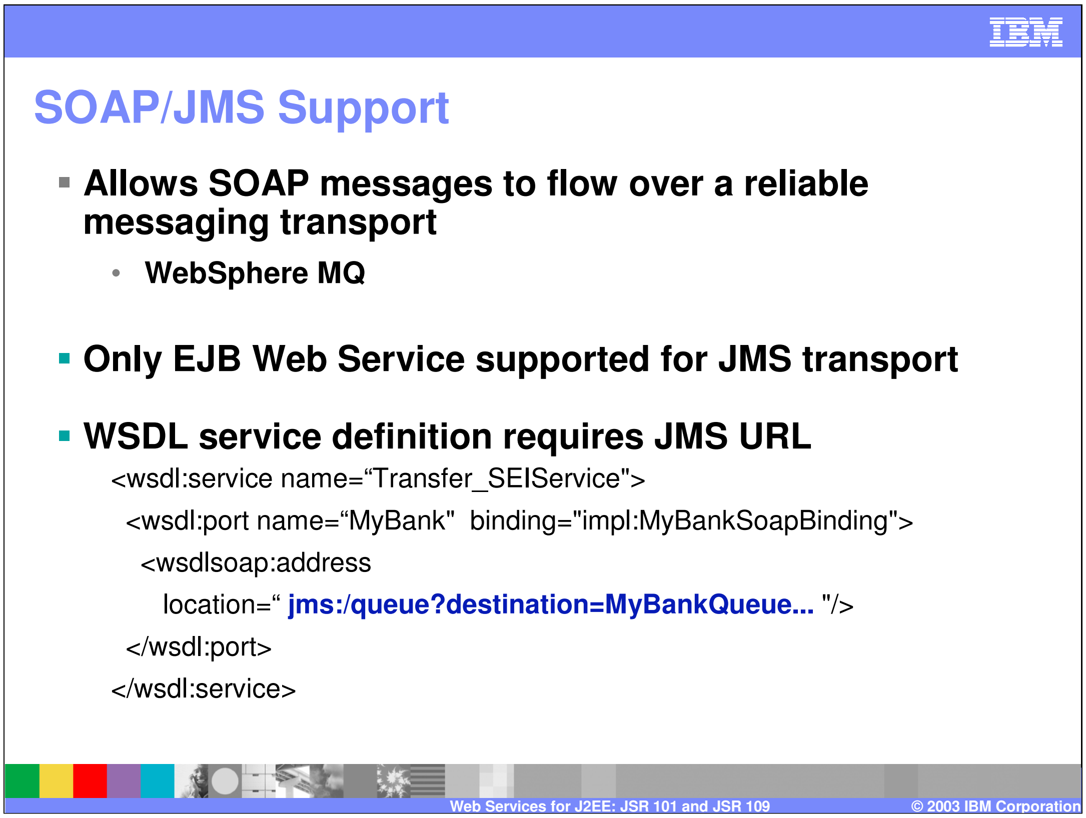
SOAP/JMS Flow - Multiple Destinations

JMS View (Topic):

One-way request to multiple consumers

Web Services for J2EE: JSR 101 and JSR 109

The publish/subscribe model is also supported by SOAP/JMS. This model is useful when you want to accommodate multiple consumers in a web services environment.



SOAP/JMS Support

- Allows SOAP messages to flow over a reliable messaging transport
  - WebSphere MQ
- Only EJB Web Service supported for JMS transport
- WSDL service definition requires JMS URL

```
<wsdl:service name=“Transfer_SEIService">
  <wsdl:port name=“MyBank"  binding="impl:MyBankSoapBinding">
    <wsdlsoap:address
       location=“ jms:/queue?destination=MyBankQueue... "/>
  </wsdl:port>
</wsdl:service>
```

Web Services for J2EE: JSR 101 and JSR 109

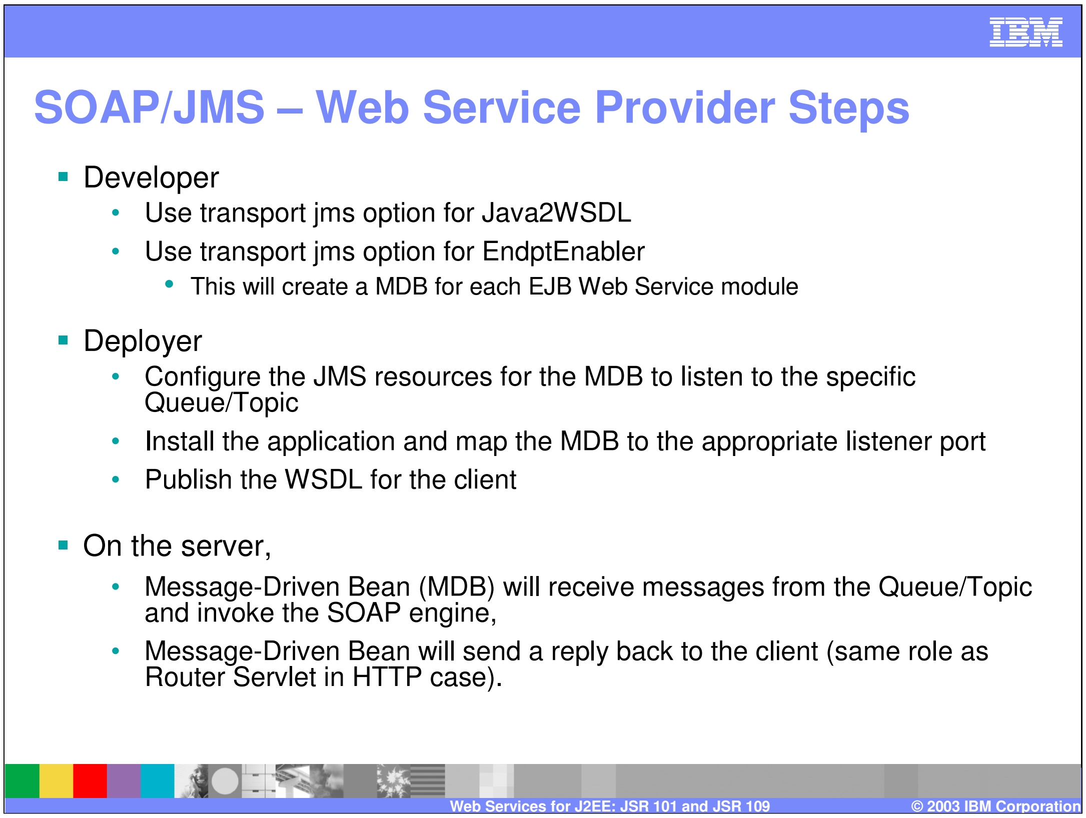
© 2003 IBM Corporation

With SOAP/JMS, SOAP messages can rely on the quality of service offered by a messaging infrastructure, such as WebSphereMQ, which guarantees message delivery, handles retries, and can be clustered for failover and workload management.

The only Web Services that can be exposed via SOAP/JMS are those that are implemented by an EJB (in other words, you can’t expose a Java Bean using SOAP/JMS).

The WSDL document that describes a SOAP/JMS web service needs to carry an indication of the JMS end point (called JMS URL). The JMS URL is in the following format:

jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF&targetService=services/Transfer_SEIService



SOAP/JMS – Web Service Provider Steps

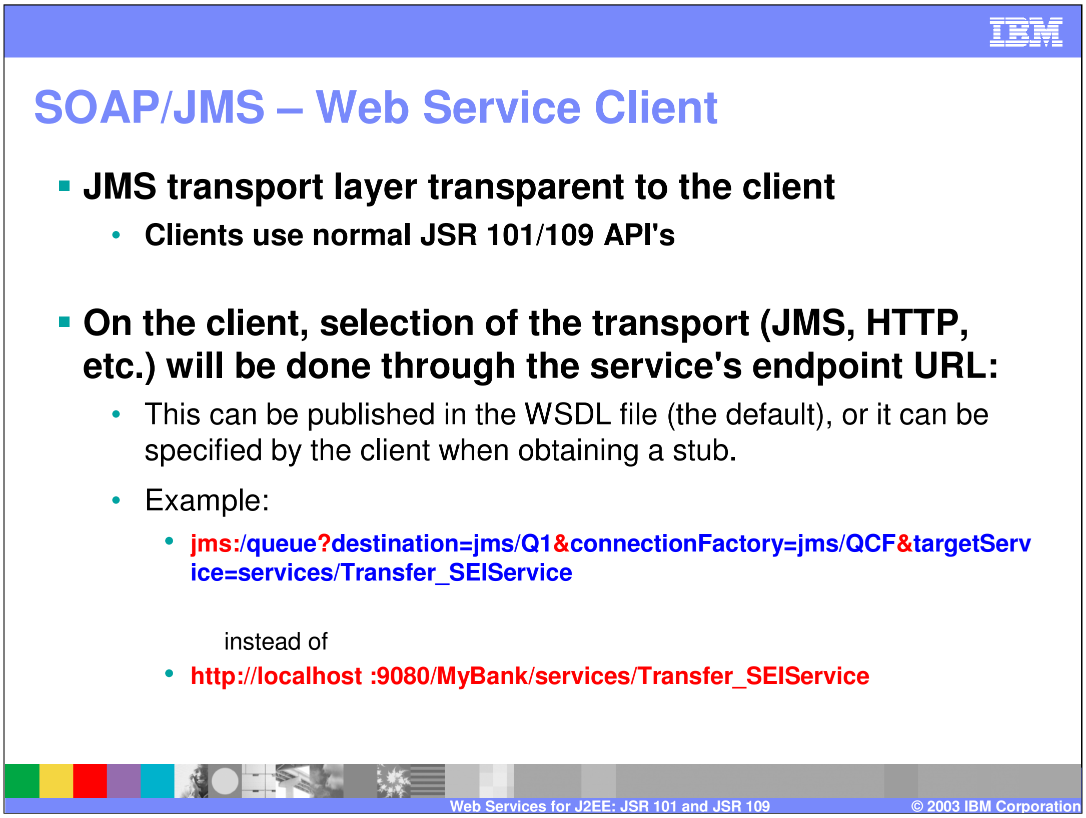
- Developer
  - Use transport jms option for Java2WSDL
  - Use transport jms option for EndptEnabler
    - This will create a MDB for each EJB Web Service module
- Deployer
  - Configure the JMS resources for the MDB to listen to the specific Queue/Topic
  - Install the application and map the MDB to the appropriate listener port
  - Publish the WSDL for the client
- On the server,
  - Message-Driven Bean (MDB) will receive messages from the Queue/Topic and invoke the SOAP engine,
  - Message-Driven Bean will send a reply back to the client (same role as Router Servlet in HTTP case).

These are the roles and the steps involved with exposing a service via SOAP/JMS.

The developer will act as in the case of SOAP/HTTP, with the exception of choosing the *jms* option for the transport. The EndPtEnabler will then create a MDB definition as a listener.

The Deployer will have to create all the necessary JMS resources to support the SOAP/JMS protocol. The queue or topic for the SOAP requests needs to be created and an associated connection factory has to be created too (typically using the console). Then a Listener Port will have to be created as well. The Listener Port (LP) references the destination and the Connection Factory (CF).

When the application is installed, the deployer will have to bind the MDB to the Listener Port. When exporting the WSDL file, the deployer will have to specify the JMS URL.



SOAP/JMS – Web Service Client

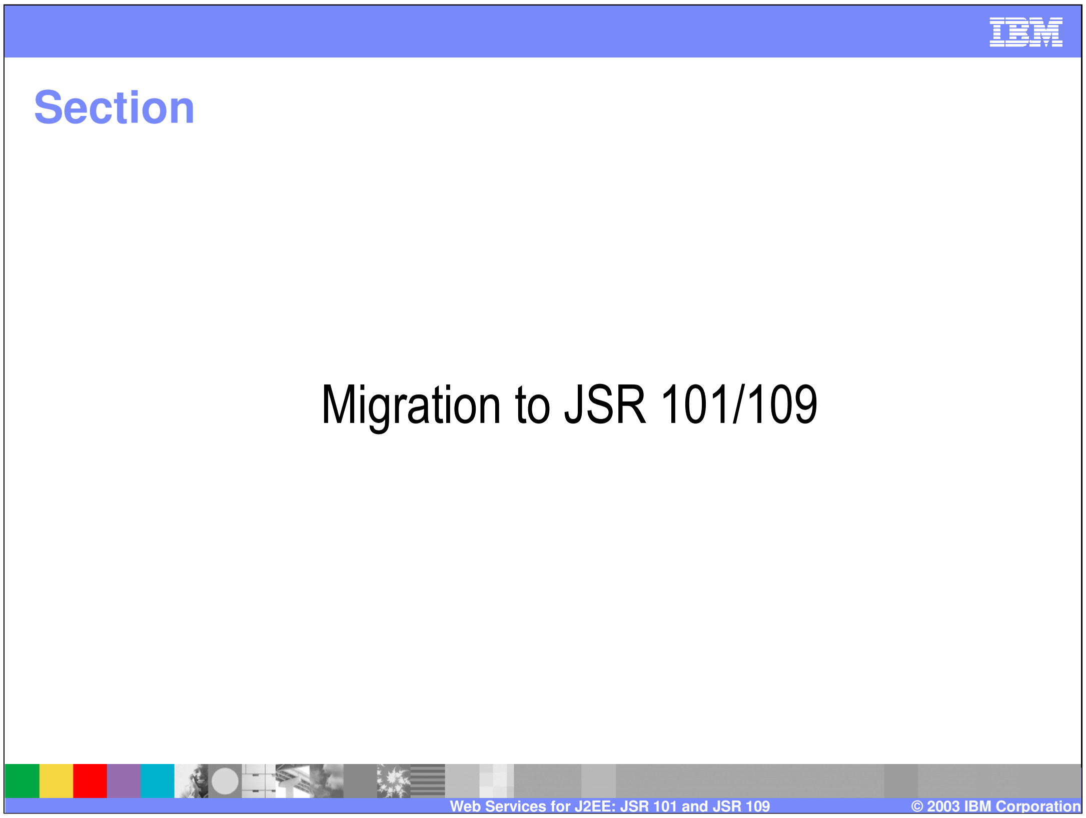
- **JMS transport layer transparent to the client**
  - **Clients use normal JSR 101/109 API's**

- **On the client, selection of the transport (JMS, HTTP, etc.) will be done through the service's endpoint URL:**
  - This can be published in the WSDL file (the default), or it can be specified by the client when obtaining a stub.
  - Example:
    - **jms:/queue?destination=jms/Q1&connectionFactory=jms/QCF&targetService=services/Transfer_SEIService**

      instead of
    - **http://localhost :9080/MyBank/services/Transfer_SEIService**

Web Services for J2EE: JSR 101 and JSR 109

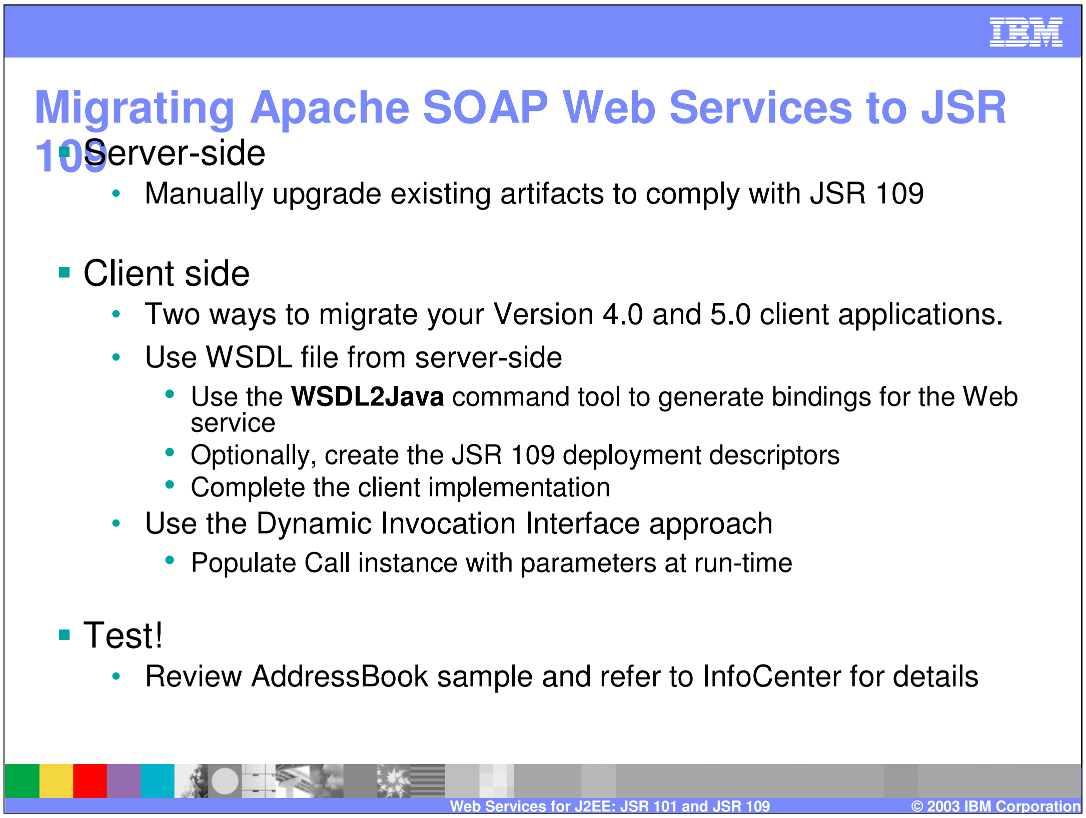
The client keeps using the standard JSR 101/109 APIs to invoke the Web Service – there is no exposure of the transport protocol being used. Under the covers, the runtime support for JSR 101/109 will translate those API calls into JMS invocations.



Section

Migration to JSR 101/109

Web Services for J2EE: JSR 101 and JSR 109



Migrating Apache SOAP Web Services to JSR 109

- Server-side
  - Manually upgrade existing artifacts to comply with JSR 109
- Client side
  - Two ways to migrate your Version 4.0 and 5.0 client applications.
  - Use WSDL file from server-side
    - Use the **WSDL2Java** command tool to generate bindings for the Web service
    - Optionally, create the JSR 109 deployment descriptors
    - Complete the client implementation
  - Use the Dynamic Invocation Interface approach
    - Populate Call instance with parameters at run-time
- Test!
  - Review AddressBook sample and refer to InfoCenter for details

Web Services for J2EE: JSR 101 and JSR 109

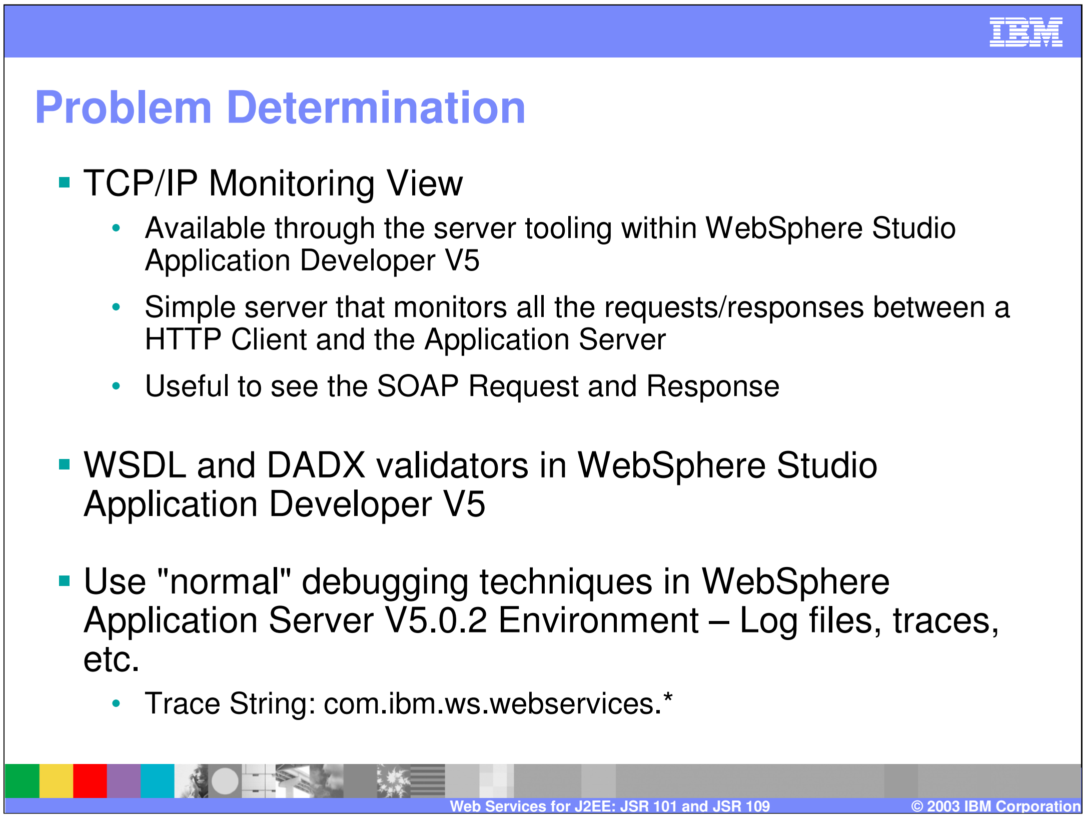
Migrating existing Apache SOAP Web Services to the new JSR 101/109 protocol is going to involve some manual intervention, as described in the chart.

The InfoCenter gives you great details on the individual development steps that are needed here. Also, the AddressBook sample shows a simple example of a JAX-RPC client implementation that you can refer to prior to migrating.

Extensive testing will be required after migration.

Section

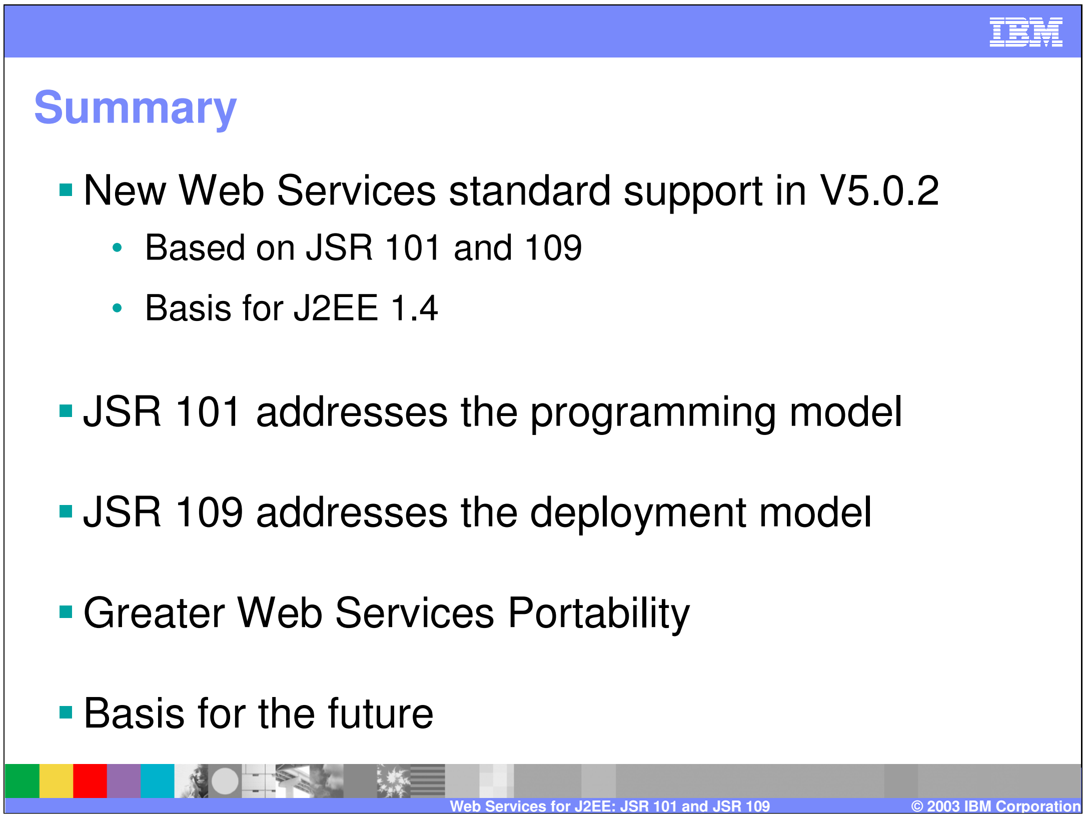
Problem Determination



Problem Determination

- TCP/IP Monitoring View
  - Available through the server tooling within WebSphere Studio Application Developer V5
  - Simple server that monitors all the requests/responses between a HTTP Client and the Application Server
  - Useful to see the SOAP Request and Response
- WSDL and DADX validators in WebSphere Studio Application Developer V5
- Use "normal" debugging techniques in WebSphere Application Server V5.0.2 Environment – Log files, traces, etc.
  - Trace String: com.ibm.ws.webservices.*

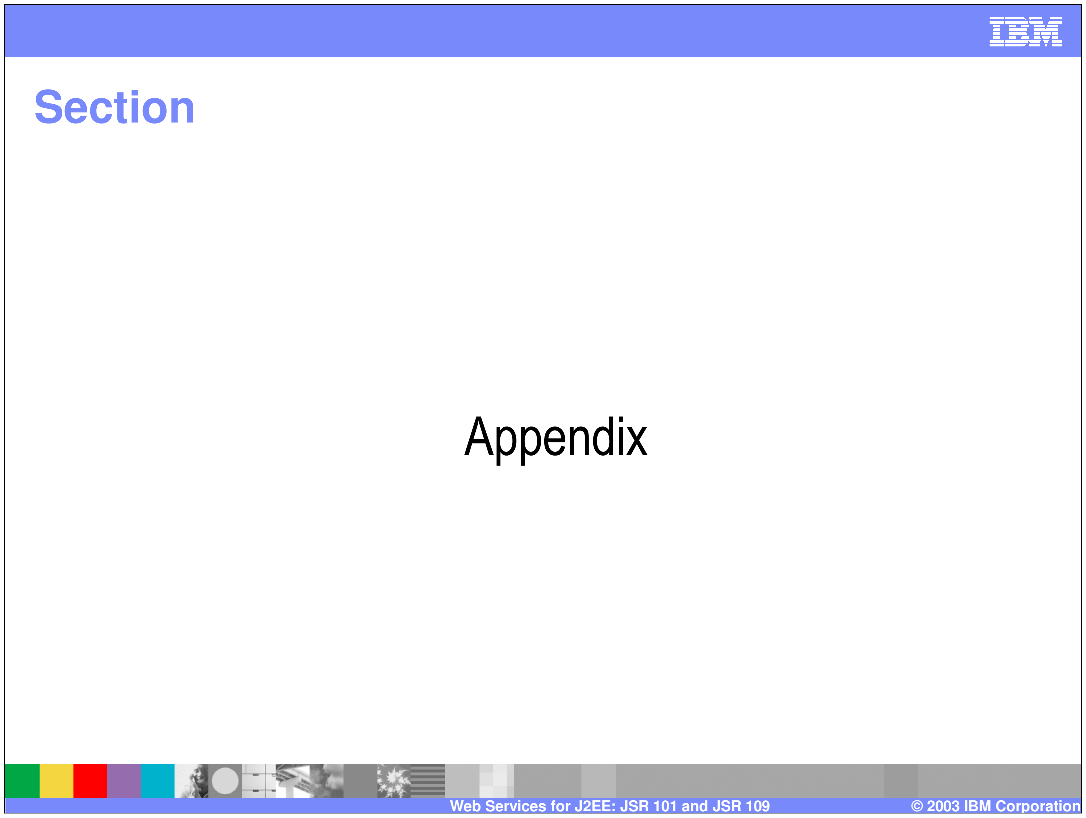
Web Services for J2EE: JSR 101 and JSR 109



Summary

- New Web Services standard support in V5.0.2
  - Based on JSR 101 and 109
  - Basis for J2EE 1.4
- JSR 101 addresses the programming model
- JSR 109 addresses the deployment model
- Greater Web Services Portability
- Basis for the future

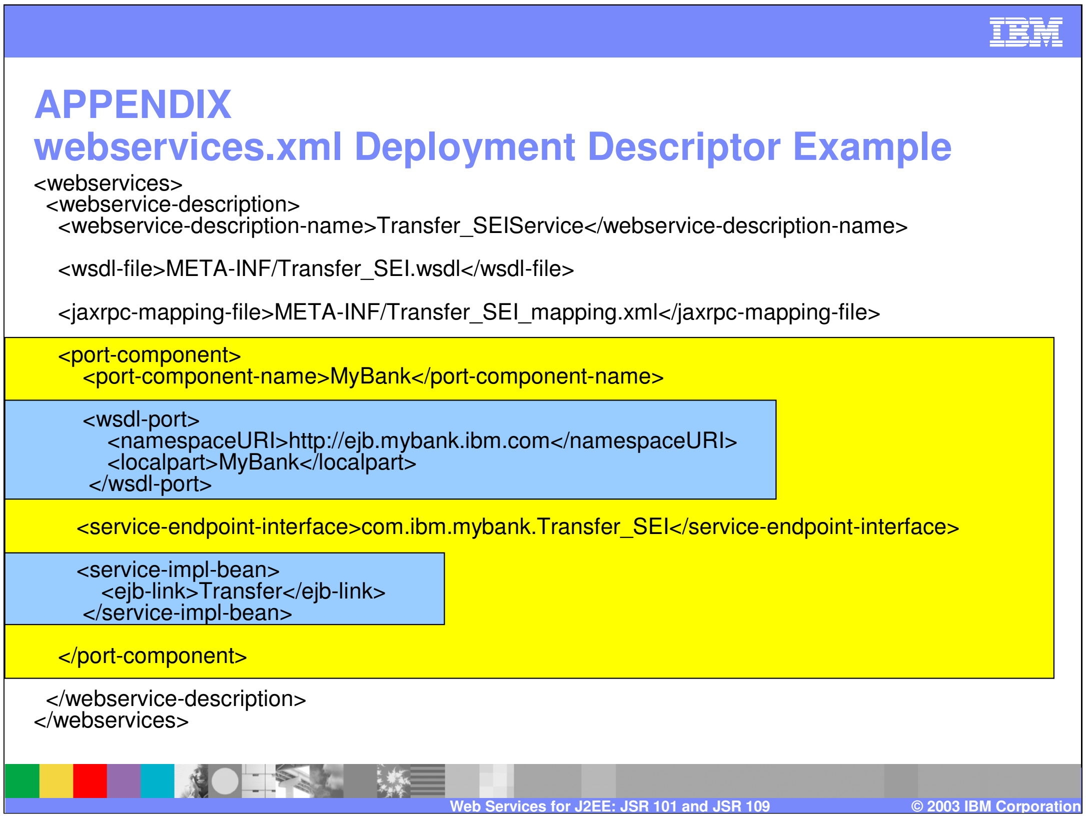
Web Services for J2EE: JSR 101 and JSR 109



Section

Appendix

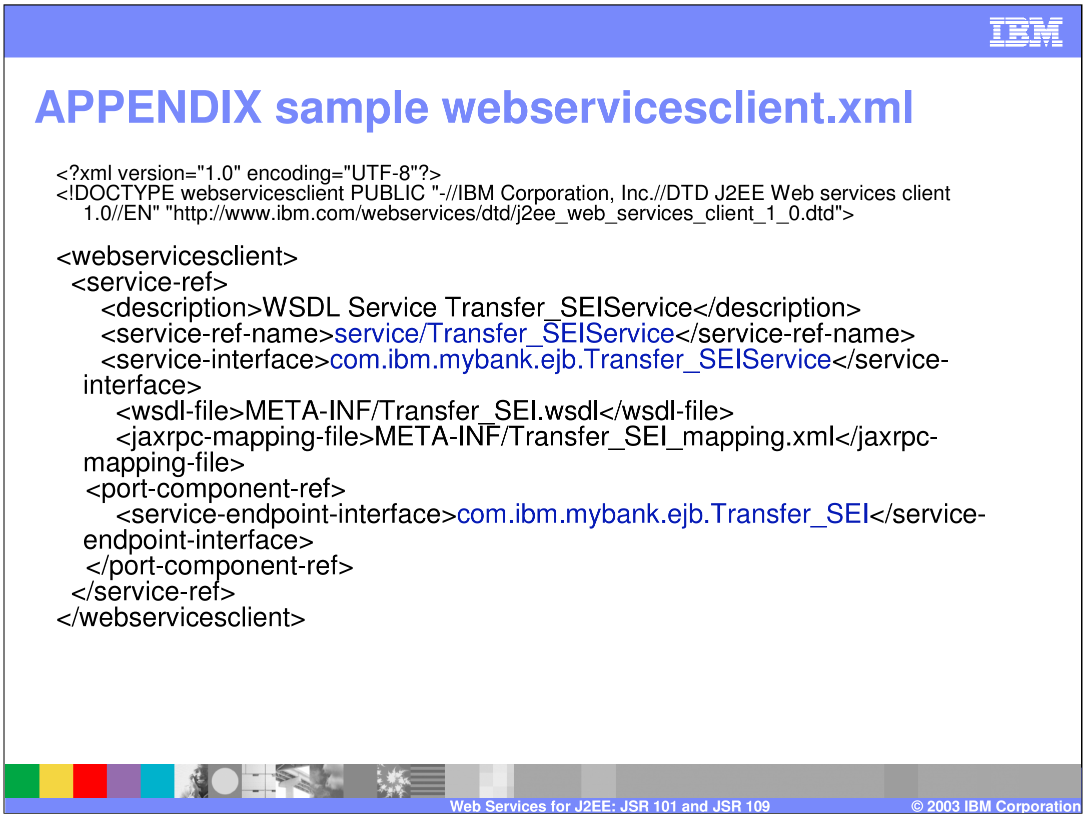
Web Services for J2EE: JSR 101 and JSR 109



APPENDIX
webservices.xml Deployment Descriptor Example

```
<webservices>
 <webservice-description>
  <webservice-description-name>Transfer_SEIService</webservice-description-name>

  <wsdl-file>META-INF/Transfer_SEI.wsdl</wsdl-file>

  <jaxrpc-mapping-file>META-INF/Transfer_SEI_mapping.xml</jaxrpc-mapping-file>

  <port-component>
      <port-component-name>MyBank</port-component-name>

      <wsdl-port>
          <namespaceURI>http://ejb.mybank.ibm.com</namespaceURI>
          <localpart>MyBank</localpart>
       </wsdl-port>

     <service-endpoint-interface>com.ibm.mybank.Transfer_SEI</service-endpoint-interface>

     <service-impl-bean>
         <ejb-link>Transfer</ejb-link>
      </service-impl-bean>

  </port-component>

 </webservice-description>
</webservices>
```

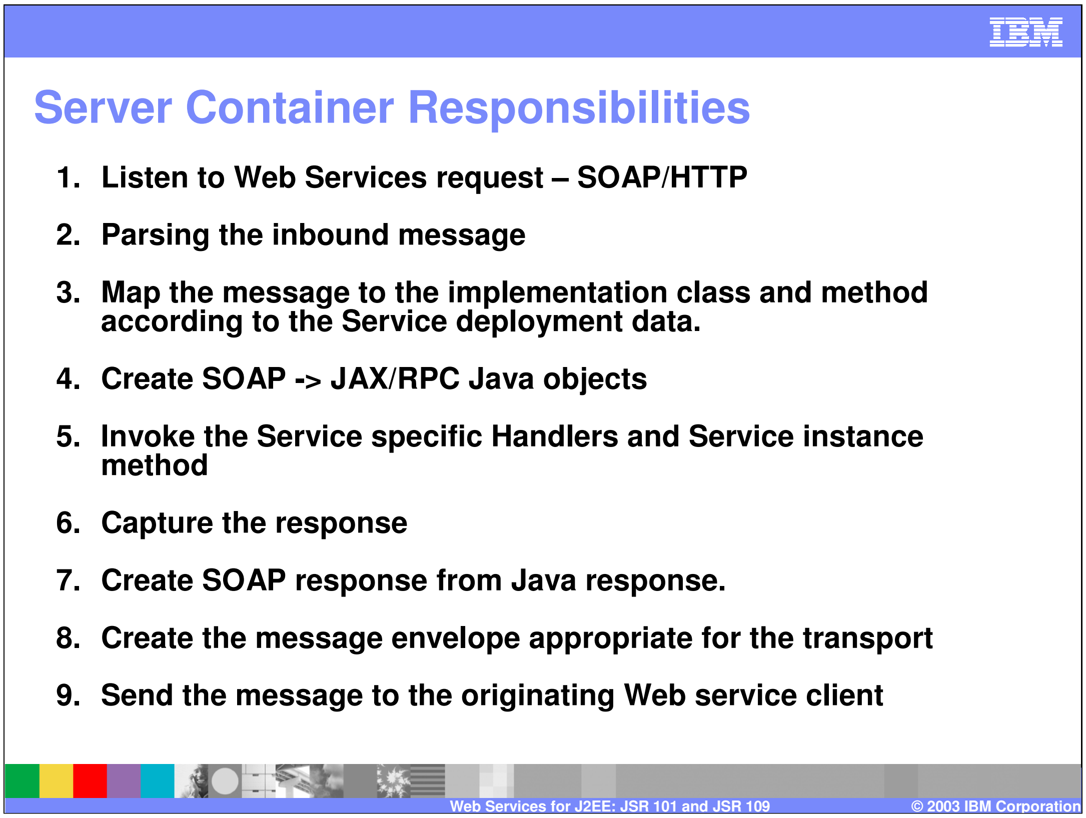
Web Services for J2EE: JSR 101 and JSR 109



APPENDIX sample webservicesclient.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE webservicesclient PUBLIC "-//IBM Corporation, Inc.//DTD J2EE Web services client
    1.0//EN" "http://www.ibm.com/webservices/dtd/j2ee_web_services_client_1_0.dtd">

<webservicesclient>
  <service-ref>
      <description>WSDL Service Transfer_SEIService</description>
      <service-ref-name>service/Transfer_SEIService</service-ref-name>
      <service-interface>com.ibm.mybank.ejb.Transfer_SEIService</service-
    interface>
        <wsdl-file>META-INF/Transfer_SEI.wsdl</wsdl-file>
        <jaxrpc-mapping-file>META-INF/Transfer_SEI_mapping.xml</jaxrpc-
    mapping-file>
    <port-component-ref>
        <service-endpoint-interface>com.ibm.mybank.ejb.Transfer_SEI</service-
    endpoint-interface>
    </port-component-ref>
  </service-ref>
</webservicesclient>
```

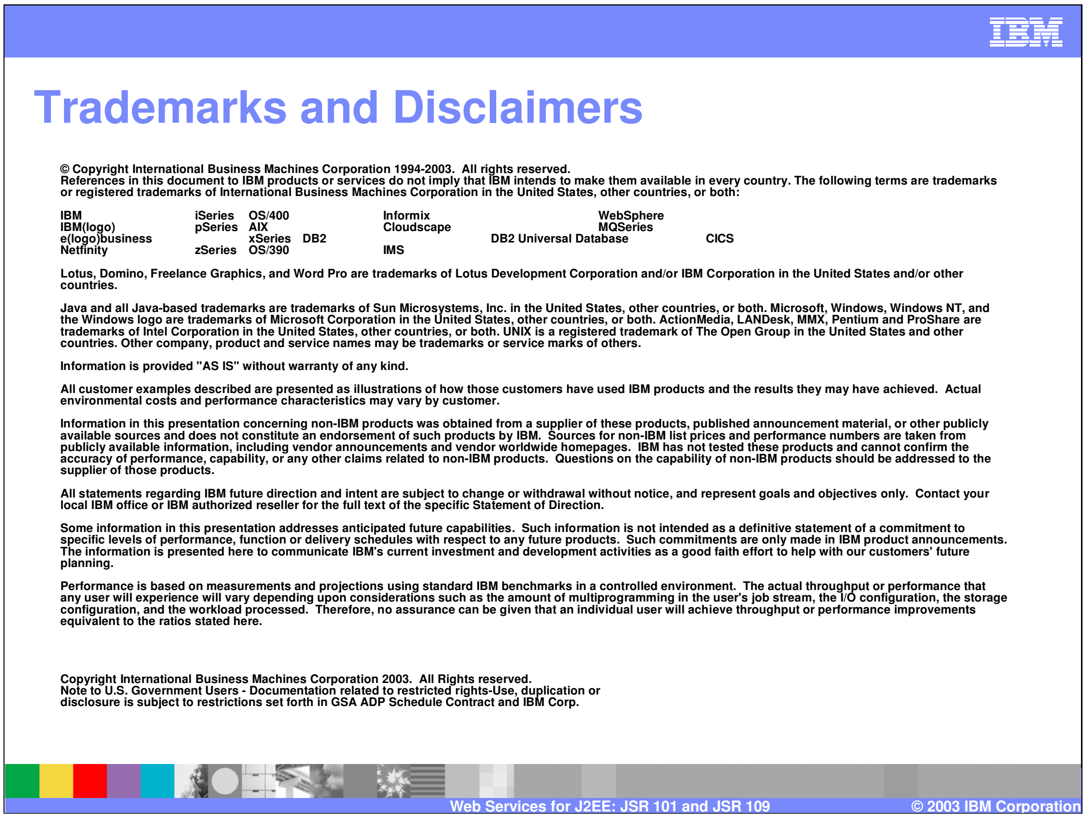
Web Services for J2EE: JSR 101 and JSR 109



Server Container Responsibilities

1. **Listen to Web Services request – SOAP/HTTP**
2. **Parsing the inbound message**
3. **Map the message to the implementation class and method according to the Service deployment data.**
4. **Create SOAP -> JAX/RPC Java objects**
5. **Invoke the Service specific Handlers and Service instance method**
6. **Capture the response**
7. **Create SOAP response from Java response.**
8. **Create the message envelope appropriate for the transport**
9. **Send the message to the originating Web service client**

Web Services for J2EE: JSR 101 and JSR 109



Trademarks and Disclaimers

References in this document to IBM products or services do not imply that IBM intends to make them available in every country. The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM iSeries OS/400 Informix WebSphere
IBM(logo) pSeries AIX Cloudscape MQSeries
e(logo)business xSeries DB2 DB2 Universal Database CICS
Netfinity zSeries OS/390 IMS

Lotus, Domino, Freelance Graphics, and Word Pro are trademarks of Lotus Development Corporation and/or IBM Corporation in the United States and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both. ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Other company, product and service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information in this presentation concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Contact your local IBM office or IBM authorized reseller for the full text of the specific Statement of Direction.

Some information in this presentation addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.