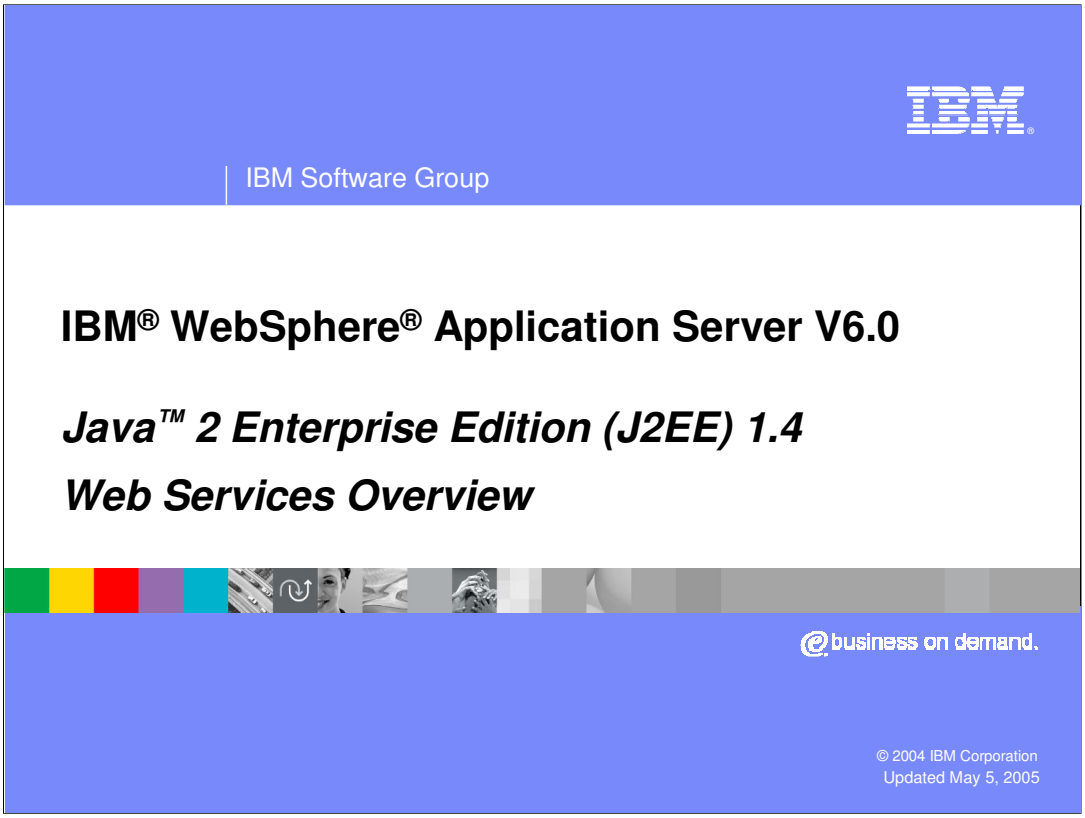
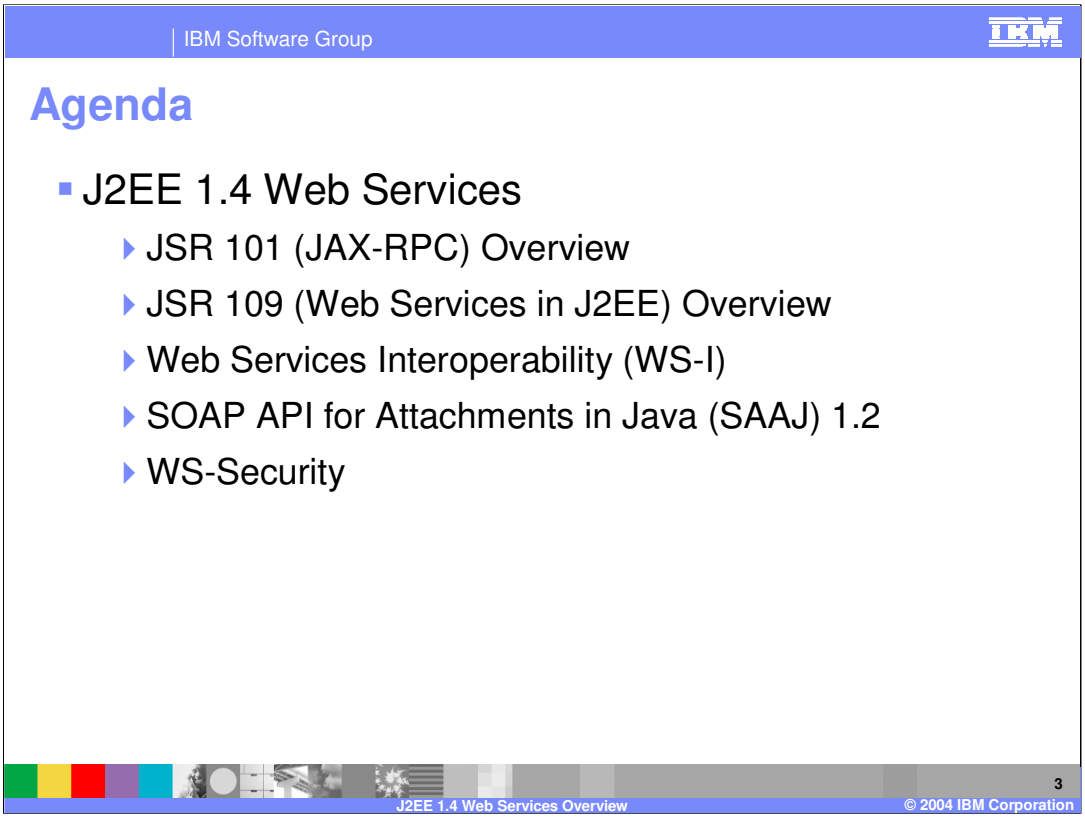


This presentation will focus on the J2EE technologies used for Web Services. A different presentation will focus on the specifics of implementing Web Services using WebSphere Application Server.

Goals

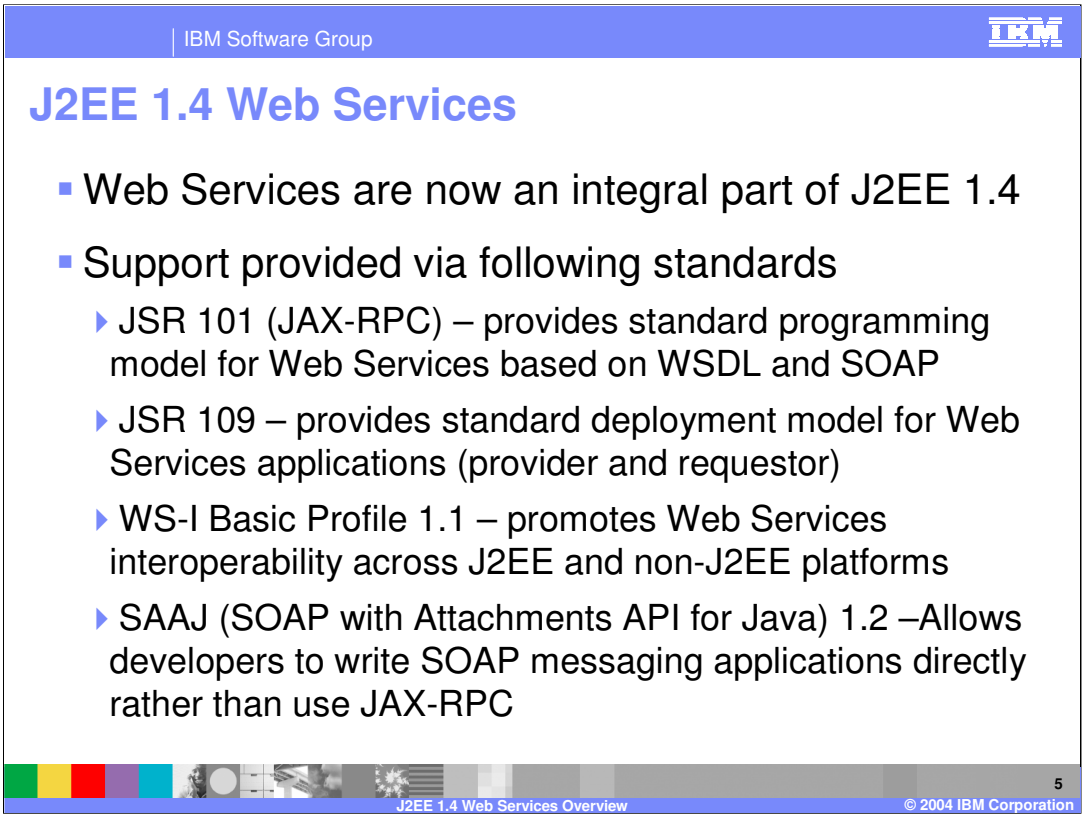
- Provide an overview of
  - J2EE 1.4 Web Services (JSR 101 and 109) implementation
  - Web Services Interoperability (WS-I)
  - WS-Security

This presentation covers the core concepts that make up Web Services. This includes the base J2EE specifications that deal with Web Services, JSR 101 and JSR 109. The presentation concludes by explaining the basic idea of Web Services Security and the benefits offered by Web Services Interoperability.



First the concepts behind JSR 101 and 109 will be discussed, which together make up the backbone of Web Services standards in J2EE. Next is a look at the standards for Web Service Interoperability. These are guidelines which, when adhered to, make your Web service interoperable with other products. A discussion of the SOAP API for Attachments in Java follows. This is an underlying API that provides methods for accessing SOAP messages, the XML format used for Web Service communications. Finally, the Web Services Security specification used to secure communication between Web Services will be covered.

This section provides an explanation of how Web Services fit within the J2EE 1.4 specification.



With the adoption of a number of Web Service specifications, Web Services have become an important part of the J2EE standard. Web Services provide a standardized means to expose processes for remote invocation. Web services support the adoption of a service oriented architecture, an architectural design by which you create a grouping of distinct services. Web Services provide the means for these services to communicate, supporting a heterogeneous environment.

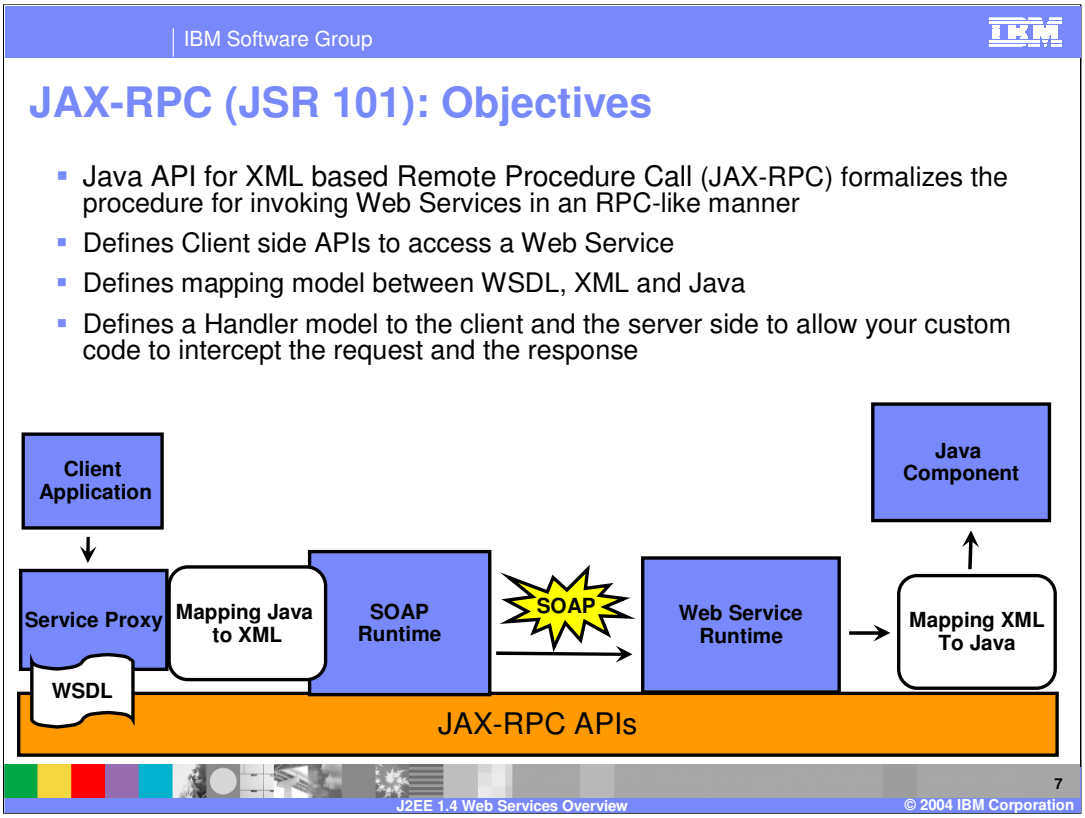
There are a number of standards currently included with the J2EE specification for Web Services support. There are many more proposed specifications for Web Services that are making their way through the community adoption process. As these are hammered out and agreed upon they will be included in the specifications.

This discussion focuses on four current standards. The JAX-RPC, or Java API for XML based Remote Procedure Calls dictates the programming model and APIs that are used for invoking and creating Web services. JSR 109 or the J2EE Web Services standard dictates how Web services operate within the J2EE environment. The WS-I standard is a more recent addition promoting interoperability between different vendors of Web services, helping to ensure the success of Web Services in heterogeneous environments. Finally, the SAAJ standard focuses on the programming model and APIs for dealing with the XML based SOAP messages.

Section

JSR 101 and 109 Overview

This section will provide more details on the core Web Services specification in J2EE, starting with JSR 101 and 109.



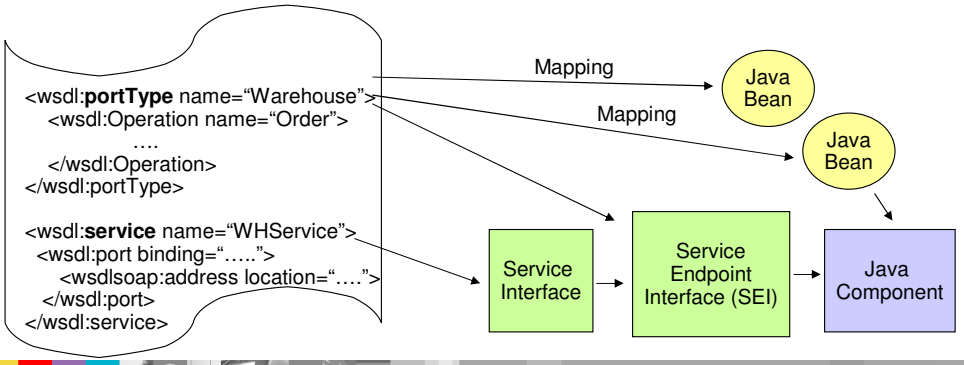
JAX-RPC defines the programming model for Web Services invocation. JAX-RPC defines client side APIs for accessing Web services. These APIs assume that a WSDL document is available at the time the service is invoked on the client. WSDL documents are used to describe the various aspects that make up a service invocation. The specification defines how information is mapped between Java, WSDL documents and XML in the SOAP message. For instance, the WSDL document contains a port type, which must be mapped to a Java artifact. The port type will be mapped to a Service Endpoint Interface, which is a particular Java interface generated for Web services.

JAX-RPC specifies a number of ways to create Web service requesting client applications. One possibility is to directly code to the JAX-RPC APIs, which would allow the client to run outside a J2EE container. This is called an unmanaged client. Alternatively, a client can run within the J2EE container as specified by JSR 109, or J2EE Web Services. This will provide a layer of abstraction and shield the client from the actual implementation details of the JAX-RPC APIs.

The specification also includes a Handler model for writing pieces of code that can intercept and modify information in the Web services message. Usually these will be written with the help of the SAAJ APIs, which are used to access the XML in the SOAP message.

JAX-RPC: Mapping between WSDL and Java

- JSR 101 defines a standardized mapping model from WSDL to 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

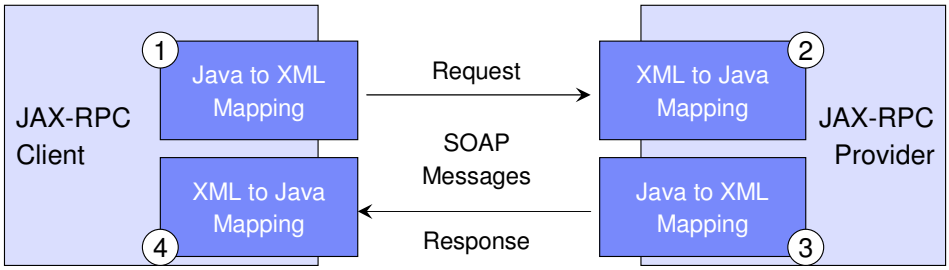


This slide provides a more detailed description of how information in the WSDL document is mapped to Java artifacts in the runtime. The port type in WSDL gets mapped to the service endpoint interface. Within the port type are a number of operations, which are mapped to Java bean implementations. There is also a service type that contains a port binding address location, which represents the address of the Web Service. This maps to a service interface that fronts the endpoint interface. These mappings are defined within the JSR 109 specification, along with other mapping rules.

In particular, JAX-RPC defines mappings between Java types and XML. JAX-RPC clients and Web service implementations deal with Java types, yet they communicate using XML in SOAP messages. That means that Java types have to be mapped to XML in order to be communicated. This process is called serialization, and the JAX-RPC standard dictates how Java types will be mapped to XML types by the runtime.

Data Mapping between XML and Java

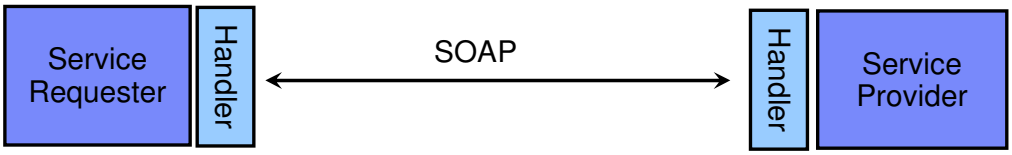
- SOAP data flows as XML elements
  - Both JAX-RPC clients and providers perform conversions to change the XML data into Java types and the reverse
    - Referred to as serialization or deserialization



This slide shows an example of the serialization and deserialization that occurs during a Web Service call. As the slide shows, during a standard request and response cycle there are four conversions that must occur. First, the client calls the target service, when the Java types understood by the client are serialized into XML types. When that is received by the Service provider, the XML message is deserialized back into Java types. The last 2 processes occur when a request message is sent back to the client.

JAX-RPC: Handlers

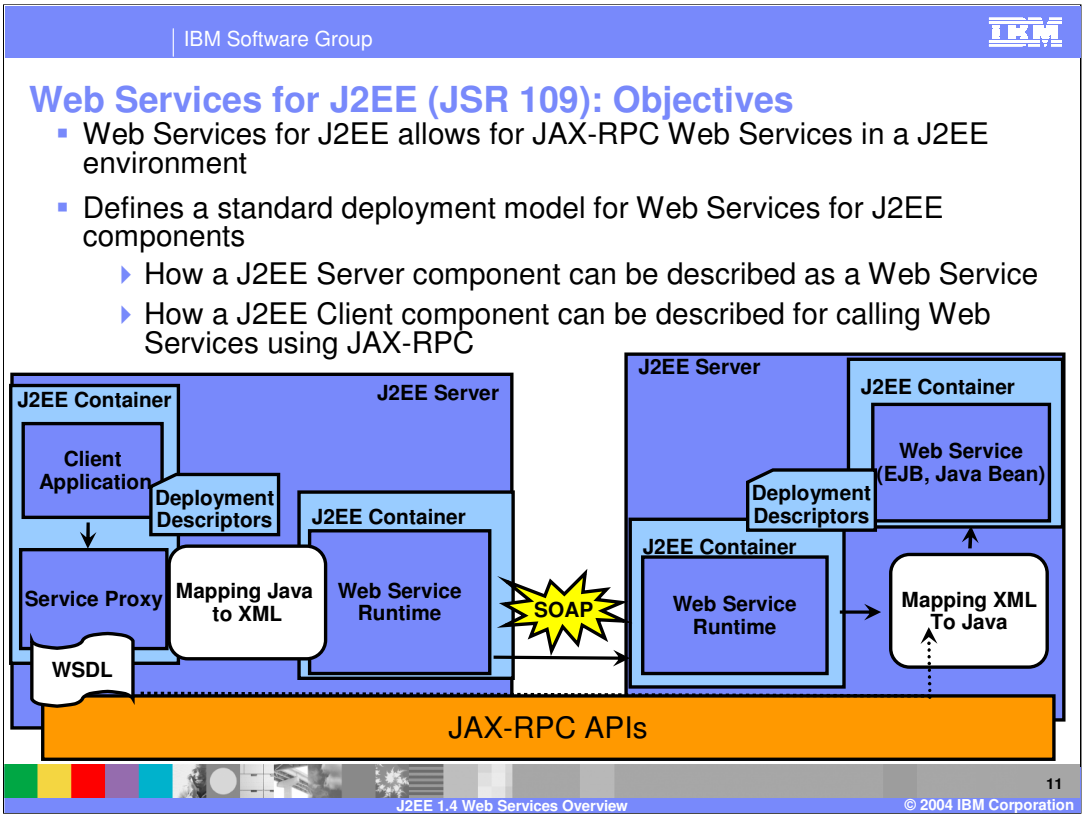
- Provide a mechanism for intercepting the SOAP message
- SOAP message handlers intercept SOAP messages in both the request and response
  - Can perform additional processing of the SOAP message
  - Can examine and potentially modify a request before it is processed by a Web Service component.
  - Can examine and potentially modify the response after the component has processed the request



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

There are some things the handlers can do and some they cannot 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. Multiple handlers can be defined for specific services on the provider side or on the client side, or both. A Handler must not change the message in any way that would cause the previously executed authorization check to execute differently.

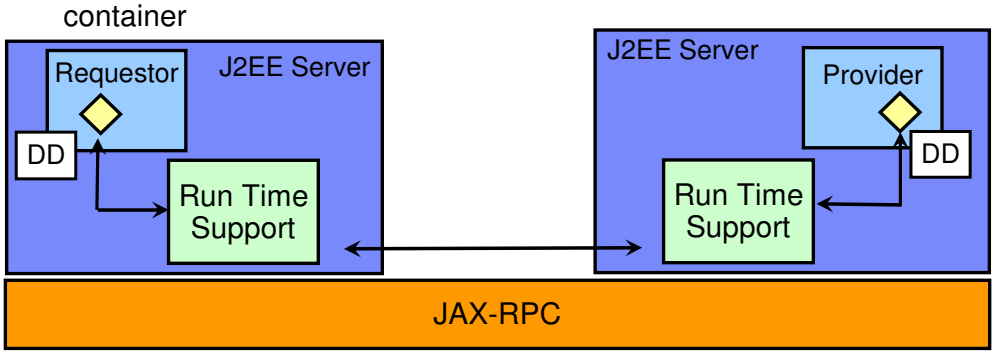
The J2EE-managed environment constrains the possible actions taken by handlers. For example, handlers cannot change the target of a request or change the operation. Handlers cannot change the message part types and number of parts. On the server, handlers can communicate with the business logic of the port component using the MessageContext. On the client side, handlers have no means of communicating with the business logic of the client.



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 application point of view so that there are commonalities among different application server vendors. In WebSphere V5.0.2, JSR 109 was fully supported. 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 the Web services to an Application Server. JSR 109 also defines special deployment descriptors 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.

Web Services Containers

- Web Services for J2EE specifies how Web Services will operate within the J2EE runtime
- Specifies container model for J2EE Web Services
  - Java classes reside in the Web container
  - EJBs must be stateless session EJBs and reside within the EJB container

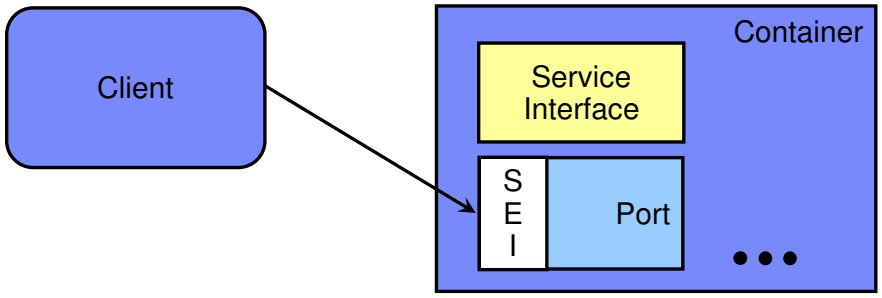


JSR 109, or the Web Services in J2EE standard works with JAX-RPC to define Web Services in J2EE environments. JSR 109 specifies the deployment descriptors that are generated for Web Services, as well as how they operate within the J2EE runtime. The deployment descriptors contain standard deployment information for services as well as security configurations.

JSR 109 also specifies the model for how Web Services operate within the J2EE container model. Depending on the type of Web Service, they will operate within either the Web container or the EJB container. Java class implementations that have been exposed as a Web Service reside within the Web container, while stateless session beans reside within the EJB container.

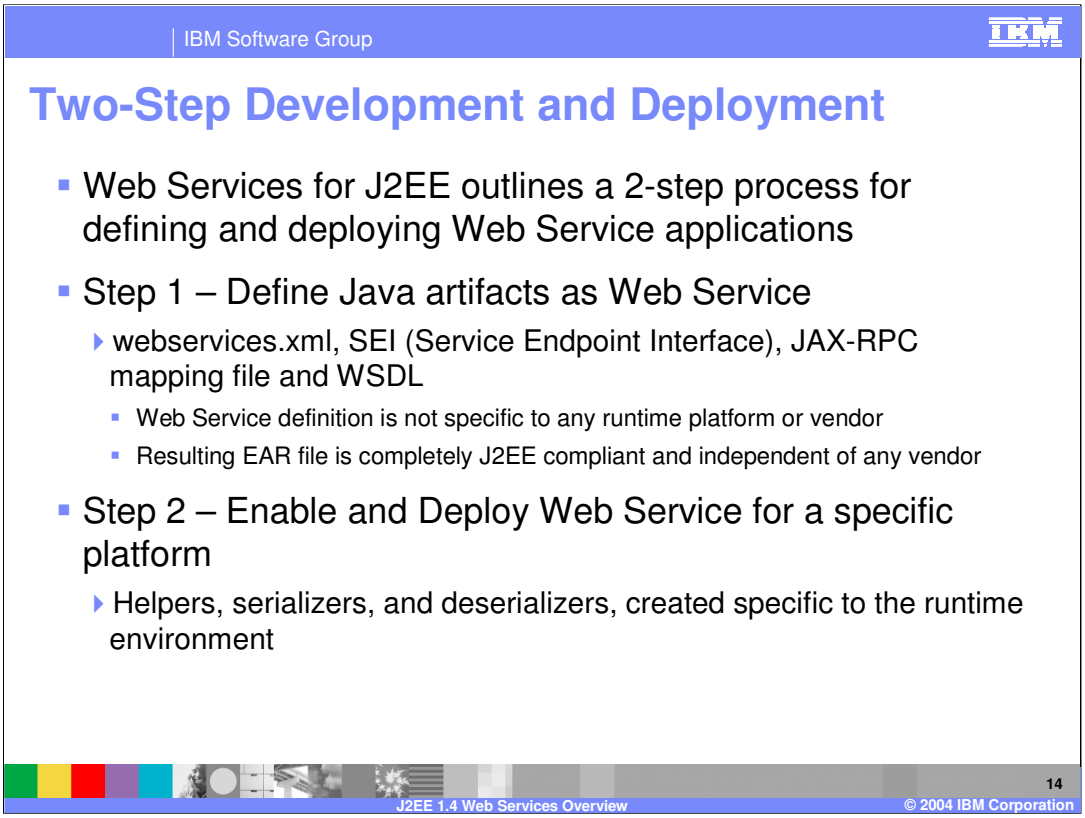
Service Interface

- Container's primary responsibility is to route SOAP messages
  - Also provides support for lifecycle management, concurrency management and security
- A Service Endpoint Interface (SEI) maps the *portType* element of the service's WSDL to a Java representation
  - Binds the WSDL representation to the port
  - Provides an implementation of the service's interface

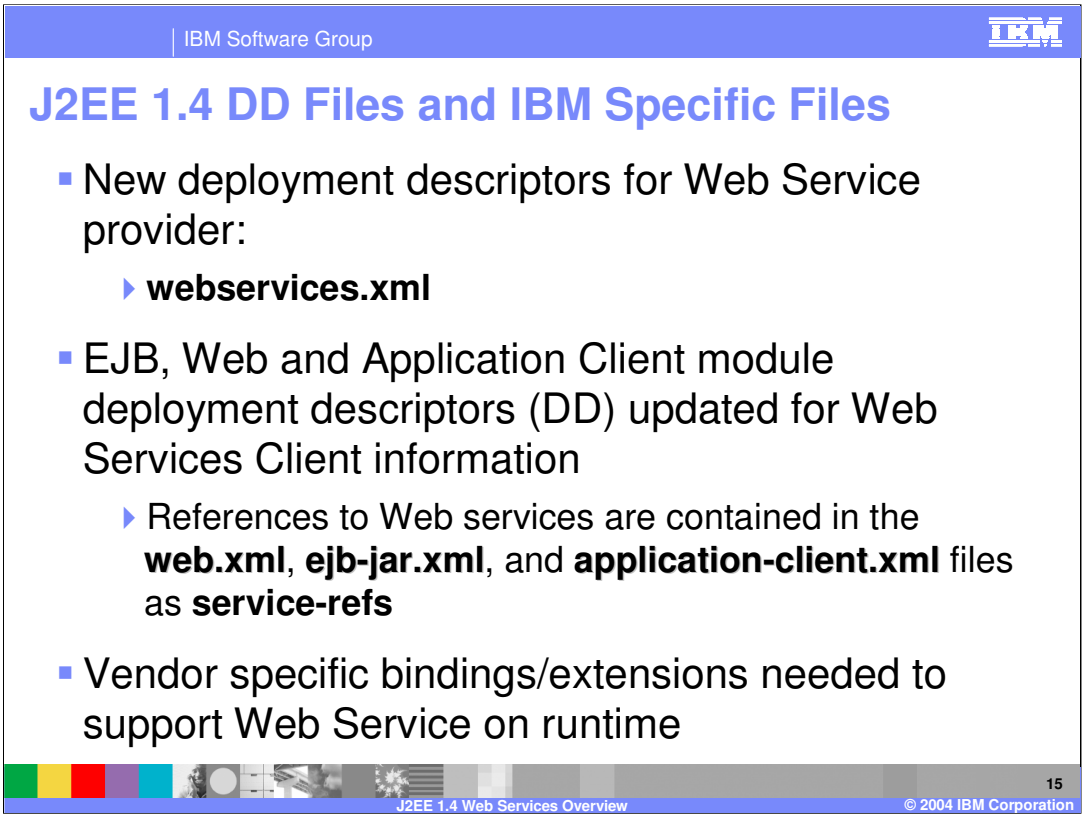


The primary job of the J2EE container, when dealing with Web services, is to route incoming SOAP messages to the appropriate Web service implementation. When a Web service is created and installed with a J2EE environment, it is fronted by an HTTP servlet which acts as a router for incoming SOAP requests. The container routes these requests to the appropriate Service Endpoint interface for the service. The container will also manage security, life cycle, and concurrency management, just as it would for any other J2EE application.

In keeping with the J2EE client programming model, a Web service client is remotable and provides local and remote transparency. The Web service port provider and the container that the port runs in define how a client sees a Web service. The client always accesses the port and is never passed a direct reference to a Web service implementation. A J2EE Web service client remains unaware of how a port operates and must concern itself only with the methods a port defines. Those methods combine to make up a Web service public interface. In addition, a client must consider access to a Web service port as stateless across service invocations. As far as the client is concerned, a port lacks a unique identity and a client has no way of determining if it communicates with identical ports across service invocations.



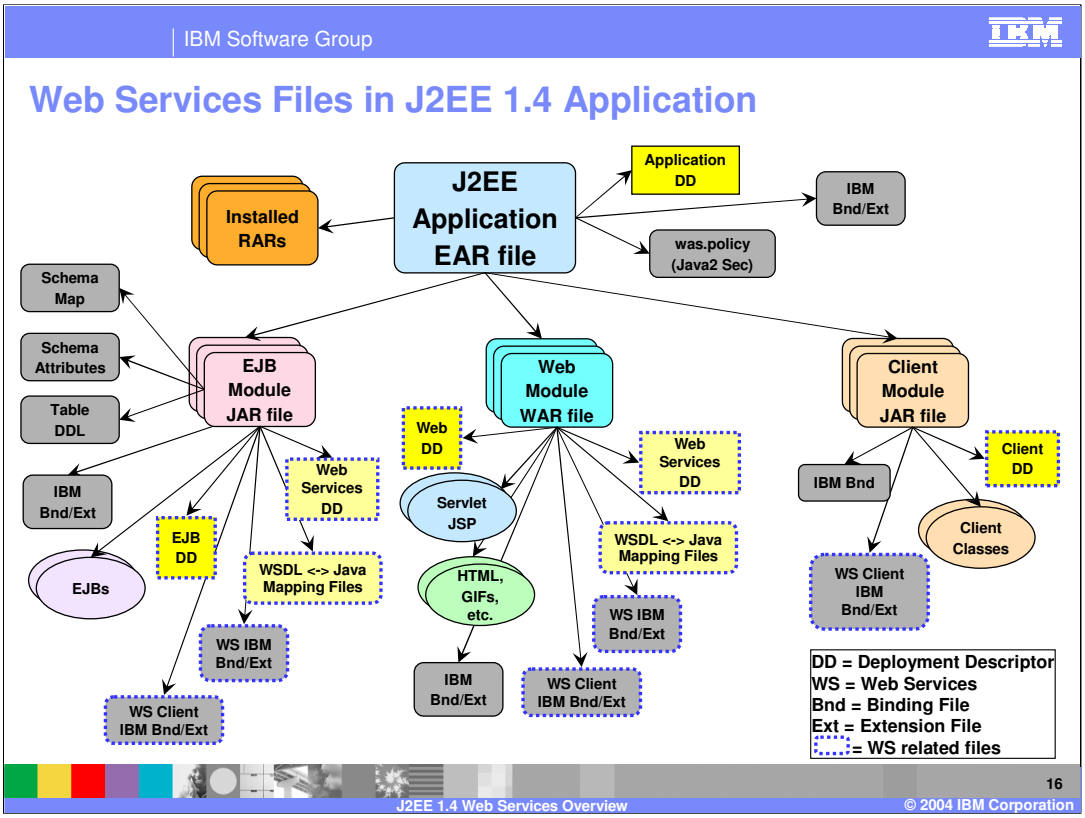
The Web Services for J2EE specification or JSR 109, describes a two-step process for creating and deploying a Web Service Application. Step one involves the process of defining Java artifacts as a Web Service. During the process the webservice.xml, service endpoint interface, and WSDL file are created. These will be packaged along with the application into a J2EE EAR, this should be independent of any vendor specific information. The second step involves deploying the Web Service application into a runtime. At this point various helpers and serializers will be created specific to the runtime environment.



A webservices.xml deployment descriptor file must be included within the EAR file that contains the service provider's implementation artifacts. In particular, that file will be included 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, a Web Application, or another EJB, must include the webservicesclient.xml deployment descriptor.

Both requester and provider applications must include a mapping metadata xml file. 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, where 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 discussed further in the Web Services Security module because these files are where the WS-Security information is included.



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 1.4 application, four if you consider the resource adapter.

The colored boxes represent the standard J2EE files. The gray boxes represent IBM specific artifacts that are needed.

For example, look at the EJB module, represented by the pink box. If you have exposed an EJB stateless session bean as a Web service provider, you will need to create a Web services deployment descriptor and the IBM extensions and bindings for that Web service provider. This can be done using IBM® Rational® Application Developer.

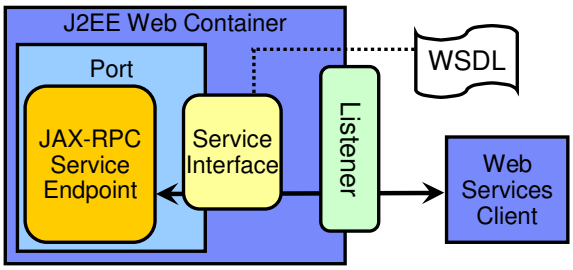
Section

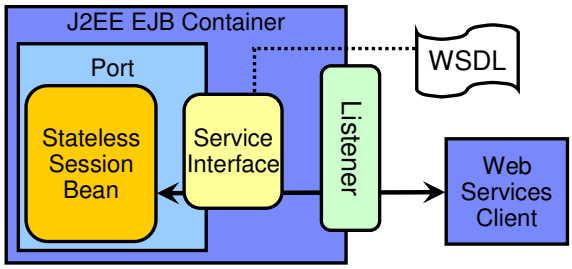
JSR 101 and 109 Client Creation

The following slides explain the steps involved with creating a J2EE compliant client.

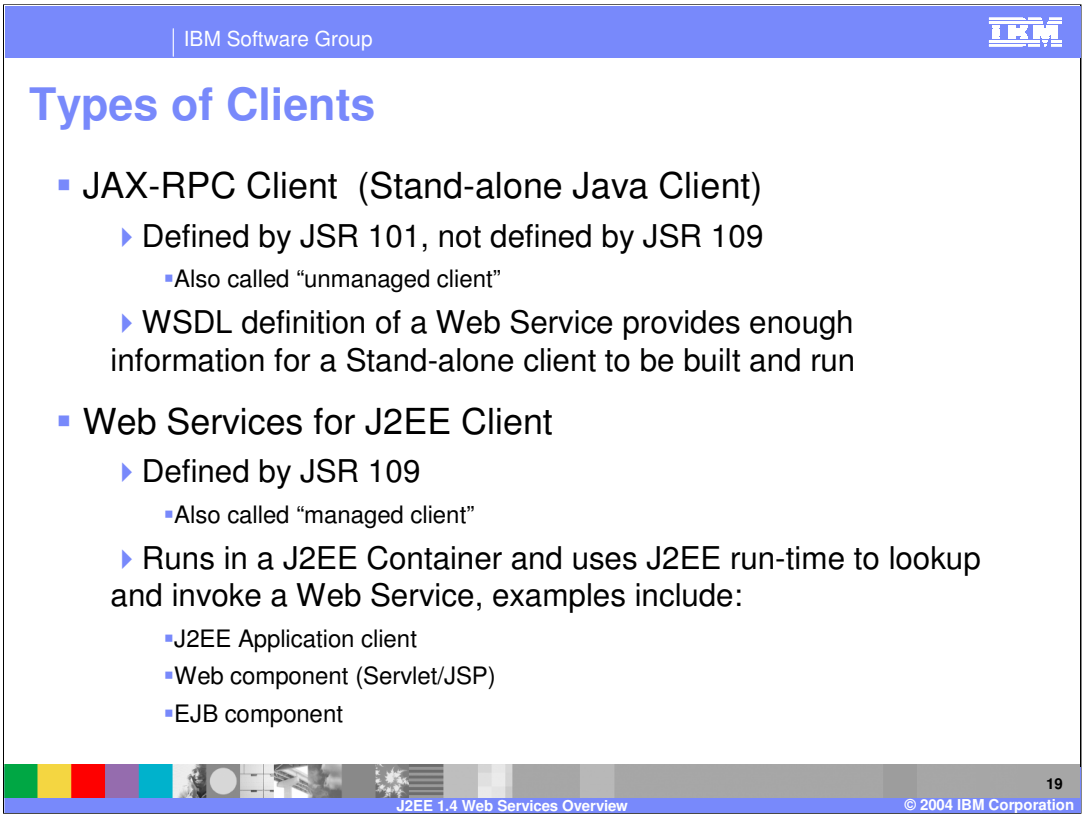
Understanding Server Side Programming Model First

- 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)





The first diagram shows a J2EE Web container, which is used when there is a Java bean as a service provider. When a Java bean is the service provider, this is referred to as a JAX-RPC service endpoint provider. The second diagram shows the J2EE EJB container, in which there is a stateless session bean as the service provider. The listeners are the components that are different. In the second diagram, the listener can be either a servlet for the SOAP/HTTP transport or a message-driven bean that is generated by the endpoint enabler if the service uses SOAP/JMS as the transport mechanism.



There are two types of basic Web Services clients.

The first type is the 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, and are therefore referred to as “unmanaged clients”. In order to invoke these clients, ensure that the JAR file is in the CLASSPATH and then use the Java command to invoke the program.

The second type is the JSR 109 client which runs inside a J2EE container. These clients are packaged as EAR files and contain components that act as service requesters. These components can be J2EE client applications, invoked using *launchClient* tool, or they can be server side components, such as servlets or Session EJBs, which 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.

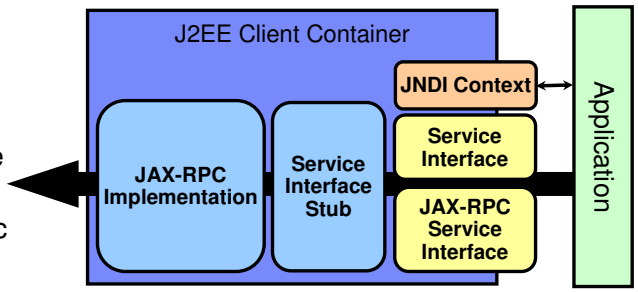
JAX-RPC Clients: Two Options

- 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 Invocation (DII) Method
    - Used when a client needs dynamic, non-stub based communication with the Web Service
- Client must always treat the Web Service implementation as stateless

The methods defined for a service interface are also categorized into two groups. The first assumes knowledge of the specific service being called, which requires knowledge of portions of the WSDL document for the target service. The other method is more dynamic and is used if the same client may connect to a number of service providers. Web Service clients must always treat the Web service implementation as stateless.

Web Services for J2EE Clients

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




```
Context ctx = new InitialContext();
javax.xml.rpc.Service sqs = ctx.lookup("java:comp/env/service/StockQuote");
com.example.StockQuoteProvider sqp =
(com.example.StockQuoteProvider)sqs.getPort(portName,
StockQuoteProvider.class);

float quotePrice = sqp.getLastTradePrice("ACME");
```


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 bindings file, it will look up a service using JNDI. It will get back a Service Endpoint Interface that can be used to transparently invoke the service, either using a stub, or less commonly by using the dynamic invocation. 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. The client cannot distinguish whether the methods are being performed locally or remotely, nor can the client distinguish how the service is implemented.

Benefits of J2EE Web Services Client

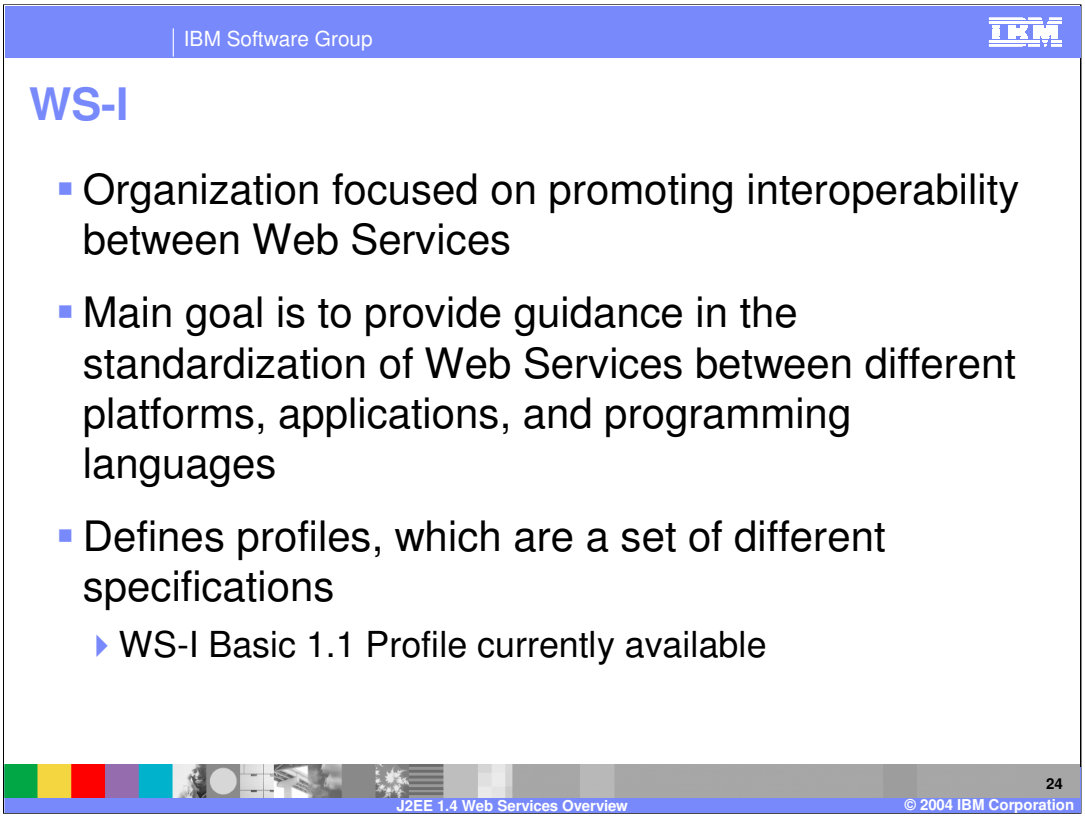
- J2EE Web Services Client programming model closely follows J2EE programming model for looking up external resources
  - Simpler for J2EE developers
- J2EE Web Service Client uses a Service Reference in the deployment descriptor to bind to Service
  - Easily configured as the service can change without changing the client code

The benefits provided by the J2EE programming model for Web Service clients are listed here. In particular, this method should be easiest for J2EE developers. A J2EE client also references a service remotely, so that the actual location of the service can change, with only minor updates needed to the J2EE client.

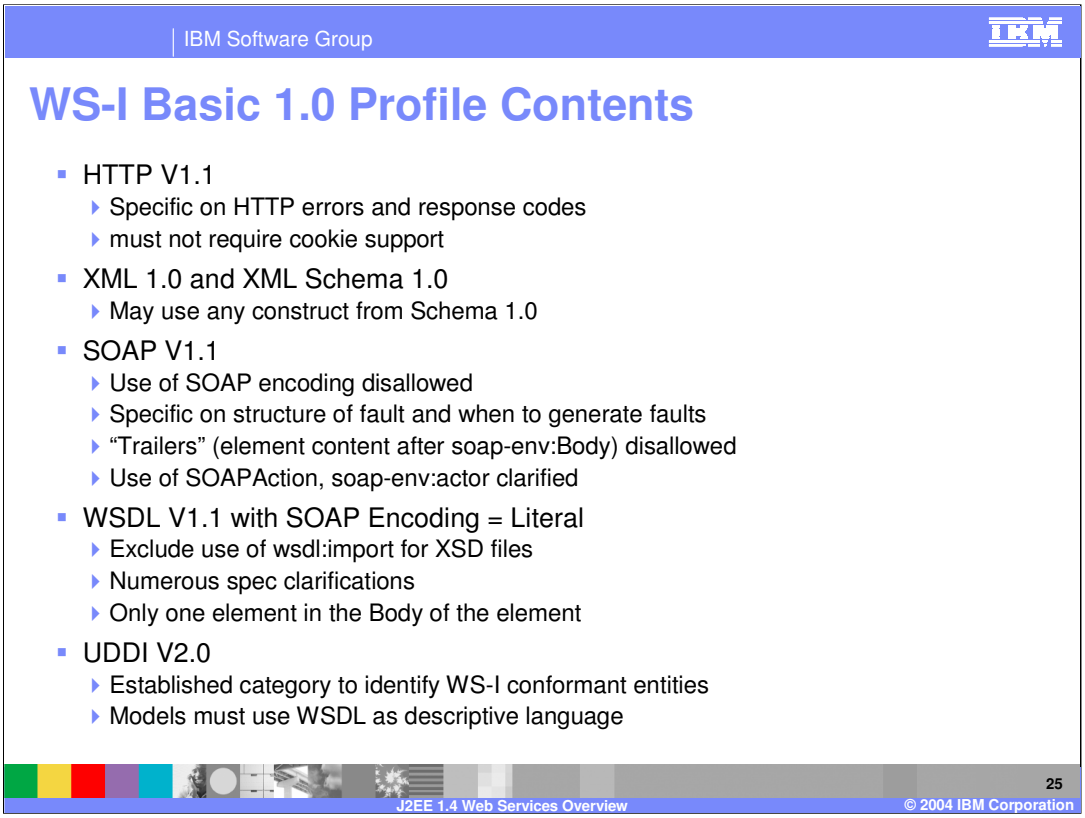
Section

WS-I
Web Services Interoperability

This section provides an explanation of the benefits of Web Services Interoperability support.



The Interoperability standard is put forward by the WSI organization, whose primary purpose is to promote interoperability between Web Services. In theory, a Web Service exposed on a .Net server should be able to be accessed no differently than a Web Service exposed on a WebSphere or other J2EE server. This organization provides standards to ensure that Web Services can interoperate across platforms and languages. They do this by creating Profiles, made up of a grouping of specific standards. So long as a Web Service is created to meet these profiles, they should interoperate with any other server or language that supports that profile. Both IBM Rational Application Developer and Eclipse provide support for creating Web services that conform to the 1.0 profile.

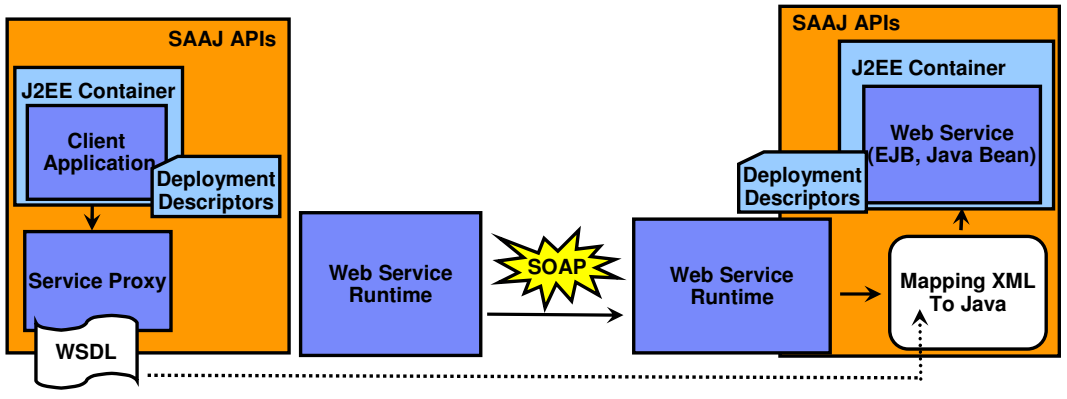


These are the various specifications that make up the WS-I Basic 1.0 Profile. Vendors will strive to become WS-I Basic 1.0 compliant rather than claim compliance with the individual specifications. By doing this, interoperability is more likely to be obtained between different Vendors. Already there is conformance on the part of the major vendors, including IBM, Microsoft™, Sun and others. The profile proscribes the underlying technologies that should be used by vendors when including support for Web Services. So the profile defines technology levels for HTTP, XML, SOAP, and WSDL. These are the technologies used in describing and communicating in Web services. The idea being that if these are kept standard across vendor implementations, interoperability will be easier to achieve.

This section provides an explanation of the Soap Attachments API for Java.

SAAJ

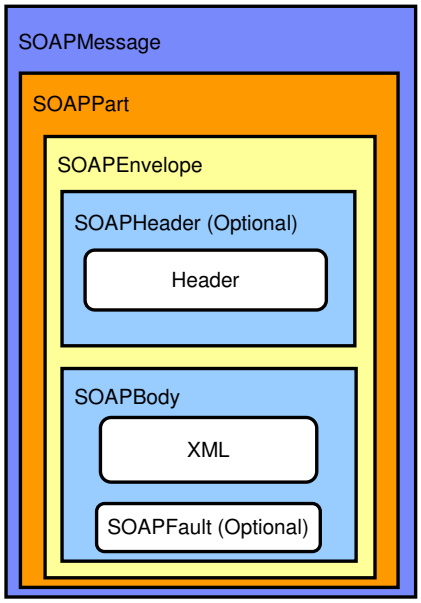
- SAAJ provides a standard set of APIs to send XML documents (including attachments) over the Internet
- Similar to JAX-RPC but requires additional effort on the client and server sides



The SAAJ APIs provide a standard means of handling SOAP messages. These APIs can be used to create, inspect, and alter SOAP messages. They can even be used to send and receive SOAP messages. Their primary use is for creating JAX-RPC handlers, as these APIs easily allow a developer to access information in the SOAP message.

SAAJ API

- Contains methods for creating and sending SOAP messages
  - Unlike JAX-RPC and J2EE Web Service clients which create the SOAP message automatically
- SOAPPart can only contain XML content
  - Does not contain attachment

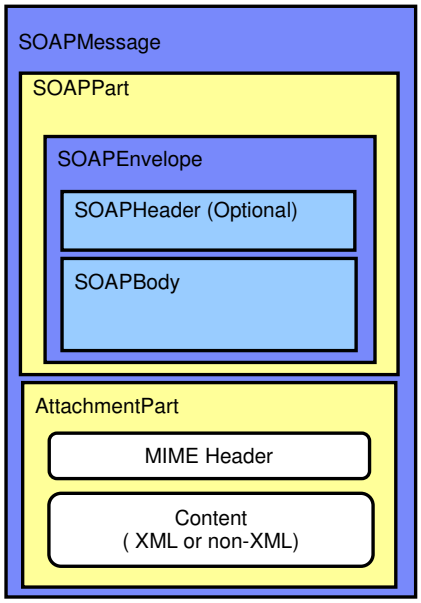


The SAAJ API, defines the format of SOAP messages. If you look at the image on the right of the screen you will see the format of the SOAP message. This contains a single SOAP part that will contain the SOAP portion of the message. The Soap part contains the SOAP envelope, which is what you will usually see referenced when describing Web Service communications. Within the SOAP envelope there is an optional header generic information. The SOAP body part is mandatory and contains the actual XML representation of the service invocation.

SAAJ also contains methods for creating and sending attachments with your soap message. Attachments con contain data in any format, and are not limited to XML, as is the SOAP message. These can be used to send data that is inappropriate to send in an XML format, such as Images. The attachments are added within the SOAP message, but outside of the SOAP part, and they must contain a MIME header that describes the format of the data.

SOAP Messages with Attachments

- A SOAPMessage may contain one or more AttachmentPart elements for attachments
- An attachment can contain data in any format
  - XML, Binary, etc…
- The MIME header is used to define the data the attachment contains

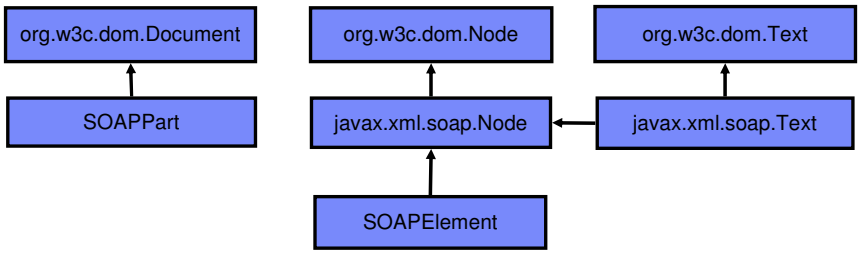


The SAAJ API provides the AttachmentPart class to represent the attachment part of a SOAP message. A SOAPMessage object automatically has a SOAPPart object and its required sub elements, but because AttachmentPart objects are optional, you must create and add them yourself.

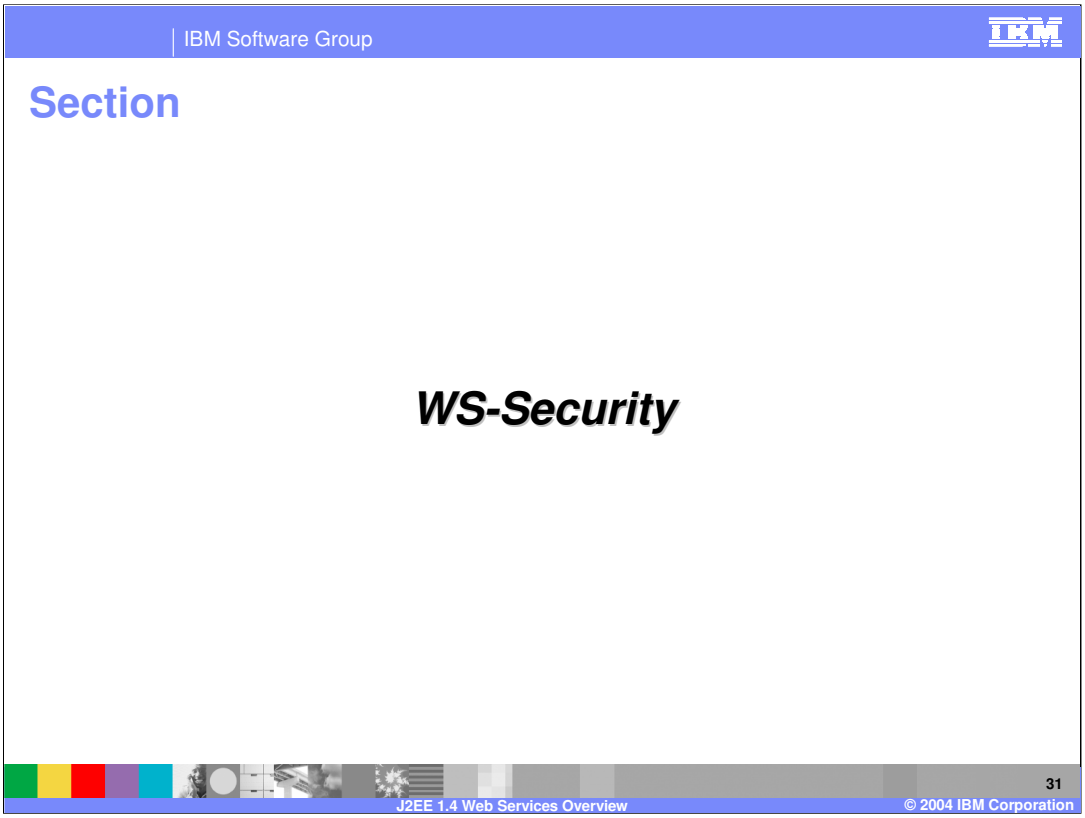
If a SOAPMessage object has one or more attachments, each AttachmentPart object must have a MIME header to indicate the type of data it contains. It may also have additional MIME headers to identify it or to give its location, which are optional but can be useful when there are multiple attachments. When a SOAPMessage object has one or more AttachmentPart objects, its SOAPPart object might or might not contain message content.

SAAJ 1.2

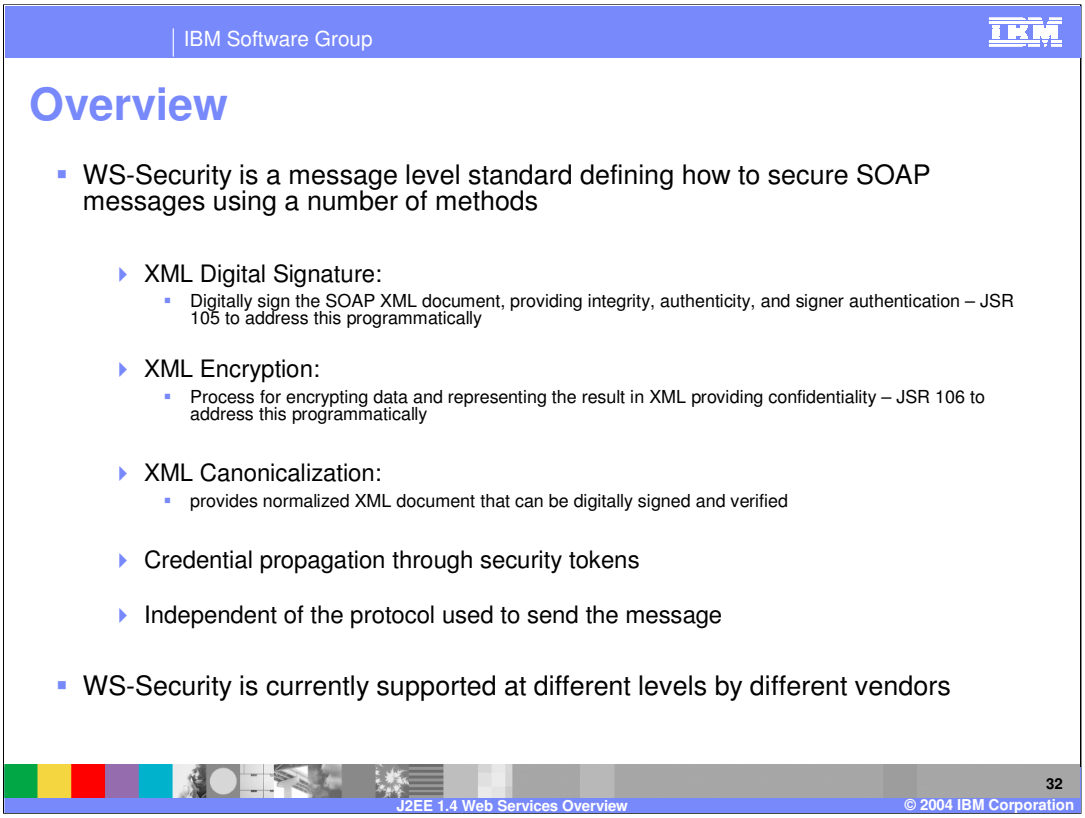
- **SAAJ 1.2 contains new APIs and binds the SAAJ APIs to Document Object Model (DOM) APIs**
  - The new APIs focus on ease of use and Java Doc support.
  - The SOAPPart of a SOAP message is now also a level 2 DOM document.
  - An object model is a mechanism for accessing a document



In J2EE 1.4 the SAAJ specification has matured to 1.2. The primary changes in SAAJ 1.2 are that the SAAJ APIs are now bound to the Document Object Model or DOM APIs. The DOM APIs provide a standard mechanism for accessing XML documents. As part of the changes the SOAP part of a SOAP message will now be considered a level 2 DOM document. These changes give more options to developers when parsing the XML of the SOAP messages.



This section provides a brief overview of WS-Security.



Web Services Security is a recently finalized standard that has been included with J2EE 1.4. In earlier versions of WebSphere Application Server, IBM provided support for WS-Security based on an early version of the specification, so some changes will occur as the product continues to conform to the new specification.

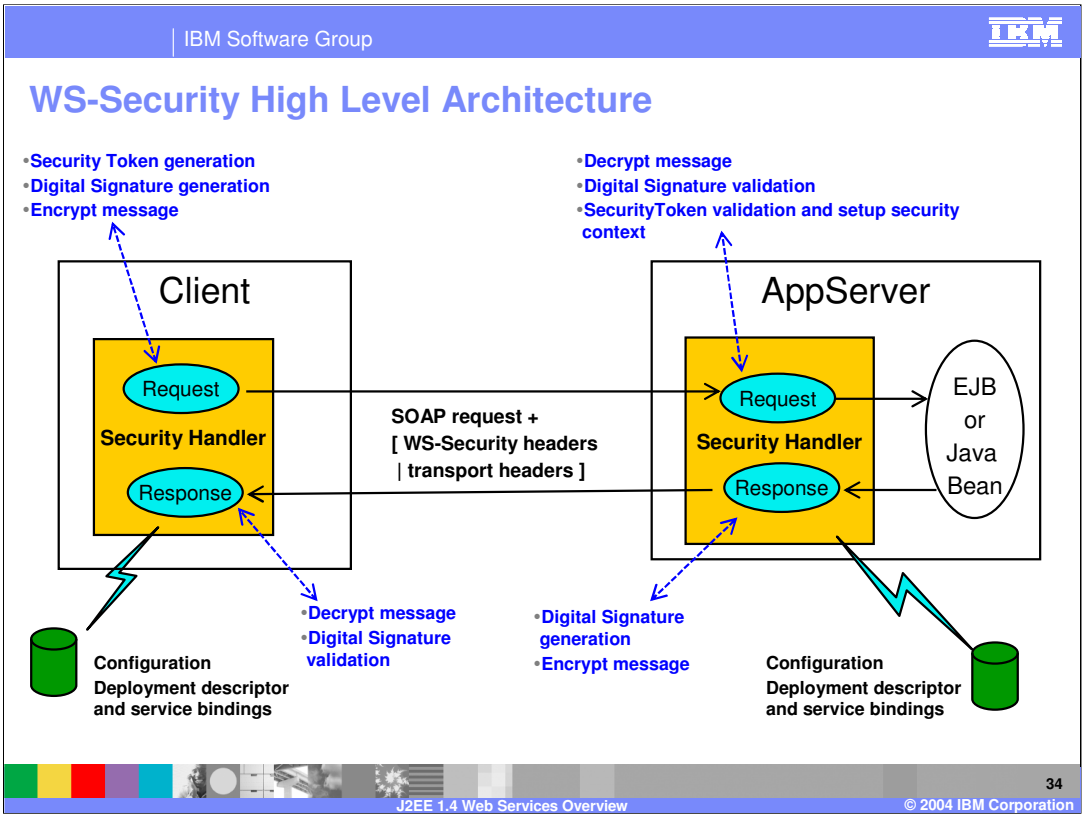
Web Services security is a message-level standard, providing security at the point of communication by including security information in the SOAP message sent by the client.

The client, based on information stored within the Web services binding and extension files inserts WS-Security information in the SOAP message. This security information can be in the form of security tokens, digital signatures, or encryption. The server, based on the information within the server-side Web services binding and extension files, will check for the security constraints in the incoming SOAP message header. The security information remains separate from the protocol used, meaning that any protocol which provides support for passing of SOAP messages can also add WS-Security information into that message.

WS-Security Implementation in WebSphere

- WS-Security is implemented as message level system handler and is registered to the Web Service runtime by the Application Server
  - Henceforth, the handlers will be referred to as the Security Handlers
- At the Client: Security handler is invoked to generate the required security headers in the SOAP message before the message is sent out to the wire
- At the Provider (Server): Security handler is called to enforce the declared security constraint in the deployment descriptor prior to dispatching the request to the Web Service Provider (EJB or Java Beans) implementation

Within the Application Server, WS-Security is implemented as a message level system handler, which is handled by the runtime. These handlers are usually referred to as the Security handlers, and are used to prepare security on a SOAP message on the client side. The handler will ensure that the appropriate security information has been added to the message before it is sent to the target provider. When the message is received by the service provider, another security handler will be the first to operate on the message. It will process the message, removing encryption and validating the security tokens contained within the message. After the security has been verified, normal operation for the Web service will continue. If the Web service makes a response to the client, security handlers will apply security as appropriate to the response.



This slide provides a high level look at the security architecture as it applies to Web Services. On the left side of the slide is a client application, making a request to the Web Service provider on the right side of the slide. The security handler associated with the request from the client application creates and applies security tokens, digital signatures, and encrypts the message. The information for security operations to perform on the message are stored within deployment descriptors and binding files associated with the client. These files are accessed by the security handler during its operations. The SOAP message with the security information is then sent to the service provider. Within the security provider's Application Server, the security handler decrypts the message and validates security tokens and digital signatures. The handler will again check information stored in the deployment descriptors and binding files to determine the appropriate security to expect on the message. If the appropriate security is not found within the message, the request will be denied. If a response is made, the security handlers will add and check security to the response message in the same way.

Summary

- Introduced J2EE 1.4 Web Services
- Discussed
  - Creating of Web Services and deploying it as a service provider
  - JSR 101/109 details
- Will discuss Web Services support in WebSphere, WS-Security and Web Services Tools in other presentations

In summary, this presentation explained the details on the J2EE specifications outlining the usage of Web Services. Various references and materials are provided on the following slides to help further explain these topics.

Resources: JSR 101, 109

- JSR 101 (JAX-RPC)
  - http://java.sun.com/xml/jaxrpc/index.html
- JSR 109
  - http://jcp.org/jsr/detail/109.jsp
  - http://www-106.ibm.com/developerworks/webservices/library/ws-jsr109/index.pdf
- Introduction to Web Services
  - http://java.sun.com/webservices/docs/ea2/tutorial/doc/IntroWS.html
- WS-I Basic Profile
  - http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html

Resources: Standards and Organizations

- http://www.w3.org/TR/SOAP/
- http://www.w3.org/TR/wsdl
- http://www.WS-I.org
- http://xml.apache.org/soap

Resources

- http://www.ibm.com/software/ad/studioappdev
- http://www.ibm.com/software/webservices
- http://www.ibm.com/developerworks/webservices
- http://www.alphaworks.ibm.com/webservices
- http://www.redbooks.ibm.com
  - SG246891 - WebSphere V5 Web Services Handbook
- http://www.eclipse.org

Trademarks, Copyrights, and Disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, 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.

Linux is a registered trademark of Linus Torvalds.

Other company, product and service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements and/or changes in the product(s) and/or program(s) described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (e.g., IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. 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.