Web Services for J2EE, Version 1.0

Please send technical comments to: wsee-spec-comments@us.ibm.com
Please send business comments to: swolfe@us.ibm.com

Specification Leads:

Jim Knutson

Heather Kreger

Final Release September 21, 2002

Specification License Agreement

IBM CORPORATION IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY ITS TERMS, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Java™ Web Services for J2EE Specification ("Specification")
Version: 1.0
Status: Final Release
Specification Lead: IBM Corporation ("Specification Lead")
Release: September 21, 2002

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of the Specification Lead and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this license and the Export Control Guidelines as set forth in the Terms of Use on the Sun's website. By viewing, downloading or otherwise copying the Specification, you agree that you have read, understood, and will comply with all of the terms and conditions set forth herein.

The Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under the Specification Lead's intellectual property rights that are essential to practice the Specification, to internally practice the Specification for the purpose of designing and developing your Java applets and applications intended to run on the Java platform or creating a clean room implementation of the Specification that: (i) includes a complete implementation of the current version of the Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of the Specification without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by the Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages or other packages defined by the Specification; (vi) satisfies all testing requirements available from the Specification Lead relating to the most recently published version of the Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any of the Specification Lead's source code or binary code materials; and (viii) does not include any of the Specification Lead's source code or binary code materials without an appropriate and separate license from the Specification Lead. The Specification contains the proprietary information of the Specification Lead and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from the Specification Lead if you fail to comply with any provision of this license. Upon termination or expiration of this license, you must cease use of or destroy the Specification.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors, the Specification Lead or the Specification Lead's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, Java Compatible, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY THE SPECIFICATION LEAD. THE SPECIFICATION LEAD MAKES NO REPRESENTATIONS OR

WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. THE SPECIFICATION LEAD MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL THE SPECIFICATION LEAD OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF THE SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. You will indemnify, hold harmless, and defend the Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide the Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant the Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Table of Contents

1 Introduction 9

1.1 Target Audience 9

1.2 Acknowledgements 9

1.3 Specification Organization 9

1.4 Document conventions. 10

2 Objectives 11

2.1.1 Client Model Goals 11

2.1.2 Service Development Goals 11

2.1.3 Service Deployment Goals 12

2.1.4 Service Publication Goals 12

2.1.5 Web Services Registry Goals 12

3 Overview 13

3.1 Web Services Architecture Overview 13

3.2 Web Service 13

3.3 Web Services for J2EE Overview 14

3.3.1 Web Service Components 15

3.3.2 Web Service Containers 15

3.4 Platform Roles 15

3.5 Portability 16

3.6 Standard Services 16

3.6.1 JAX-RPC 16

3.7 Interoperability 16

3.8 Scope 16

3.8.1 Scope 16

3.8.2 Not in Scope 17

3.9 Web Service Client View 17

3.10 Web Service Server View 18

4 Client Programming Model 19

4.1 Concepts 19

4.2 Specification 20

4.2.1 Service Lookup 20

4.2.2 Service Interface 21

4.2.3 Port Stub and Dynamic Proxy 23

4.2.4 JAX_RPC Properties 24

4.2.5 JAX-RPC Custom Serializers / Deserializers 24

4.2.6 Packaging 24

5 Server Programming Model 25

5.1 Goals 25

5.2 Concepts 25

5.3 Port Component Model Specification 26

5.3.1 Service Endpoint Interface 27

5.3.2 Service Implementation Bean 27

5.3.3 Service Implementation Bean Life Cycle 29

5.3.4 JAX-RPC Custom Serializers / Deserializers 30

5.4 Packaging 30

5.4.1 EJB Module Packaging 31

5.4.2 Web App Module Packaging 31

5.4.3 Assembly within an EAR file 31

5.5 Transactions 31

5.6 Container Provider Responsibilities 31

6 Handlers 33

6.1 Concepts 33

6.2 Specification 34

6.2.1 Scenarios 34
6.2.2 Programming Model 34
6.2.3 Developer Responsibilities 36
6.2.4 Container Provider Responsibilities 37
6.3 Packaging 37
6.4 Object Interaction Diagrams 38
6.4.1 Client Web service method access 39
6.4.2 EJB Web service method invocation 39
7 Deployment Descriptors 41
7.1 Web Services Deployment Descriptor 41
7.1.1 Overview 41
7.1.2 Developer responsibilities 41
7.1.3 Assembler responsibilities 42
7.1.4 Deployer responsibilities 42
7.1.5 Web Services Deployment Descriptor DTD 43
7.2 Web Service Client Deployment Descriptor 47
7.2.1 Overview 47
7.2.2 Developer responsibilities 48
7.2.3 Assembler responsibilities 48
7.2.4 Deployer responsibilities 49
7.2.5 Web Services Client Deployment Descriptor DTD 49
7.3 JAX-RPC Mapping Deployment Descriptor 53
7.3.1 Overview 54
7.3.2 Developer responsibilities 54
7.3.3 Assembler responsibilities 55
7.3.4 Deployer responsibilities 56
7.3.5 JAX-RPC Mapping DTD 56

8 Deployment 64
8.1 Overview 64
8.2 Container Provider requirements 65
8.2.1 Deployment artifacts 65
8.2.2 Generate Web Service Implementation classes 65
8.2.3 Generate deployed WSDL 66
8.2.4 Publishing the deployed WSDL 66
8.2.5 Service and Generated Service Interface implementation 66
8.2.6 Static stub generation 67
8.2.7 Type mappings 67
8.2.8 Mapping requirements 67
8.2.9 Deployment failure conditions 67
8.3 Deployer responsibilities 67
9 Security 69
9.1 Concepts 69
9.1.1 Authentication 69
9.1.2 Authorization 71
9.1.3 Integrity and Confidentiality 71
9.1.4 Audit 71
9.1.5 Non-Repudiation 71
9.2 Goals 71
9.2.1 Assumptions 72
9.3 Specification 72
9.3.1 Authentication 72
9.3.2 Authorization 72
9.3.3 Integrity and Confidentiality 72
Appendix A. Relationship to other Java Standards 73

Appendix B. References 74

Appendix C. Revision History 75

Appendix C.1. Version 0.95 Final Draft 75

Appendix C.2. Version 0.94 75

Appendix C.3. Version 0.93 75

Appendix C.4. Version 0.92 75

Appendix C.5. Version 0.8 76

Appendix C.6. Version 0.7 76

Appendix C.7. Version 0.6 76

Appendix C.8. Version 0.5 76

Appendix C.9. Version 0.4 Expert Group Draft 76

1 Introduction

This specification defines the Web Services for J2EE architecture. This is a service architecture that leverages the J2EE component architecture to provide a client and server programming model which is portable and interoperable across application servers, provides a scalable secure environment, and yet is familiar to J2EE developers.

1.1 Target Audience

This specification is intended to be used by:

- J2EE Vendors implementing support for Web services compliant with this specification
- Developers of Web service implementations to be deployed into J2EE application servers
- Developers of Web service clients to be deployed into J2EE application servers
- Developers of Web service clients that access Web service implementations deployed into J2EE application servers

This specification assumes that the reader is familiar with the J2EE platform and specifications. It also assumes that the reader is familiar with Web services, specifically the JAX-RPC Specification and WSDL documents.

1.2 Acknowledgements

This specification's origins are based on the vision of Donald F. Ferguson, IBM Fellow. It has been refined by an industry wide expert group. The expert group included active representation from the following companies: IBM, Sun, Oracle, BEA, Sonic Software, SAP, HP, Silverstream, IONA. We would like to thank those companies along with other members of the JSR 109 expert group: EDS, Macromedia, Interwoven, Rational Software, Developmentor, interKeel, Borland, Cisco Systems, ATG, WebGain, Sybase, Motorola, and WebMethods. We particularly appreciate the amount of input and support provided by Mark Hapner (Sun).

The JSR 109 expert group had to coordinate with other JSR expert groups in order to define a consistent programming model for Web Service for J2EE. We would like to especially thank Rahul Sharma and the JSR 101 (JAX-RPC) expert group, Farukh Najmi and the JSR 093 (JAX-R) expert group, and Linda G. DeMichiel and the JSR 153 (EJB 2.1) expert group.

1.3 Specification Organization

The next two chapters of this specification outline the requirements and conceptual architecture for Web services support in J2EE environments. Each of the major integration points for Web services in J2EE, the client model, the server model, the deployment model, WSDL bindings, and security have their own chapter. Each of these chapters consists of two topics: Concepts and Specification. The concepts section discusses how Web services are used, issues, considerations, and the scenarios that are supported. The specification section is normative and defines what implementers of this specification must support.

1.4 Document conventions.

In the interest of consistency, this specification follows the document conventions used by the Enterprise JavaBeans specification.

The regular Times font is used for information that is prescriptive by this specification.

The italic Times font is used for paragraphs that contain descriptive information, such as notes describing typical use, or notes clarifying the text with prescriptive specification.

```
The Courier font is used for code examples.
```

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

2 Objectives

This section lists the high level objectives of this specification.

- Build on the evolving industry standards for Web services, specifically WSDL 1.1 and SOAP 1.1.
- Leverage existing J2EE technology.
- Ensure that programmers may implement and package Web services that properly deploy onto application servers that comply with J2EE and this specification.
- Ensure that vendor implementations of this specification inter-operate, i.e. a Web service client on one vendor's implementation must be able to interact with Web services executing on another vendors implementation.
- Define the minimal set of new concepts, interfaces, file formats, etc. necessary to support Web services within J2EE.
- Clearly and succinctly define functions that J2EE application server vendors need to provide.
- Define the roles that this specification requires, the functions they perform and their mapping to J2EE platform roles. Define the functions that a Web Services for J2EE product provider must provide to support these roles.
- Support a simple model for defining a new Web service and deploying this into a J2EE application server.

The relation of this specification to J2EE 1.4 is defined in Appendix A.

2.1.1 Client Model Goals

The client programming model should be conformant and compatible with the client programming model defined by JAX-RPC.

Additional goals for the client programming model are to ensure that:

- Programmers can implement Web services client applications conforming to this specification that may reside in a J2EE container (e.g. an EJB that uses a Web service), or a J2EE Client Container can call a Web service running in a Web Services for J2EE container.
- Client applications conforming to this specification can call any SOAP 1.1 based Web service through the HTTP 1.1 or HTTPS SOAP Bindings.
- Programmers using other client environments such as Java 2 Standard Edition environment can call a Web service running in a Web Services for J2EE container. Programmers using languages other than Java must be able to implement SOAP 1.1 compliant applications that can use Web services conforming to this specification. Support the Client Development Scenarios described in Chapter 4.
- Client developers must not have to be aware of how the service implementation is realized.
- Java 2 Micro Edition clients, defined by JSR 172, should be able to interoperate using the transport standards declared within WSDL and the JAX-RPC runtime with Web Services for J2EE applications.

2.1.2 Service Development Goals

The service development model defines how web service implementations are to be developed and deployed into existing J2EE containers and includes the following specific goals:

- How the Web service has been implemented should be transparent to the Web service client. A client should not have to know if the Web service has been deployed in a J2EE or non-J2EE environment.
- Because the Web service implementation must be deployed in a J2EE container, the class implementing the service must conform to some defined requirements to ensure that it does not compromise the integrity of the application server.
- JAX-RPC defines three server side run time categories, J2SE based JAX-RPC Runtime, Servlet Container Based JAX-RPC Runtime, and J2EE Container Based JAX-RPC Runtime. This specification defines the J2EE container based (Web and EJB) runtime such that it is consistent with the Servlet Container based model defined by the JAX-RPC specification.
- Support mapping and dispatching SOAP 1.1 requests to methods on J2EE Stateless Session Beans.
- Support mapping and dispatching SOAP 1.1 requests to methods on JAX-RPC Service Endpoint classes in the Web Container.

2.1.3 Service Deployment Goals

- Web service deployment is declarative. We do this through extending the J2EE model for deployment descriptors and EAR file format. These changes are minimized, however.
- The Web service deployment is supportable on top of existing J2EE 1.3 environments.
- Deployment requires that a service be representable by WSDL. Deployment requires a WSDL file. The deployment of Web services must support:
  - those who wish to deploy a Web service as the focus of the deployment
  - those who wish to expose existing, deployed J2EE components as a Web service

2.1.4 Service Publication Goals

- Service deployment may publish the WSDL to the appropriate service registry, repository (if required by the Web service), File, or URL.
- If a Web service needs to be published by the deployment tools, all of the data required to perform the publication must be provided in the deployment package or during the deployment process.
- If any publication to UDDI is performed, the WSDL must also be made available at a URL.

2.1.5 Web Services Registry Goals

The Web services registry API and programming model is out of the scope of this specification. The Web service implementation, Web service client, or Web service deployment tool may use any registry API including JAX-R. JAX-R does not support WSDL publication directly. It does support interaction with UDDI directories. UDDI.org specifies how to publish a WSDL described service to a UDDI directory.

This specification defines the service publication responsibilities of the deployment tool.

Service definition discovery (finding the WSDL to be implemented) during development or deployment of a service implementation is not defined.

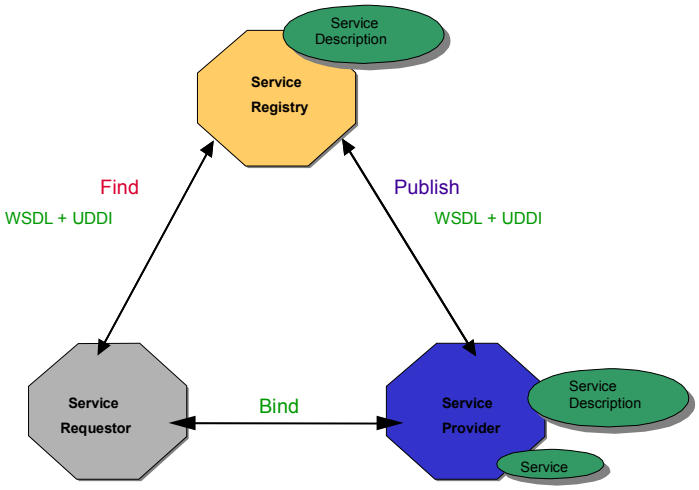
Service discovery during development, deployment, or runtime of service clients is not defined.

3 Overview

This chapter provides an overview of Web services in general and how Web Services for J2EE fits into the J2EE platform.

3.1 Web Services Architecture Overview

Web services is a service oriented architecture which allows for creating an abstract definition of a service, providing a concrete implementation of a service, publishing and finding a service, service instance selection, and interoperable service use. In general a Web service implementation and client use may be decoupled in a variety of ways. Client and server implementations can be decoupled in programming model. Concrete implementations may be decoupled in logic and transport.



• Figure 1 Service oriented architecture

The service provider defines an abstract service description using the Web Services Description Language (WSDL). A concrete Service is then created from the abstract service description yielding a concrete service description in WSDL. The concrete service description can then be published to a registry such as Universal Description, Discovery and Integration (UDDI). A service requestor can use a registry to locate a service description and from that service description select and use a concrete implementation of the service.

The abstract service description is defined in a WSDL document as a PortType. A concrete Service instance is defined by the combination of a PortType, transport & encoding binding and an address as a WSDL port. Sets of ports are aggregated into a WSDL service.

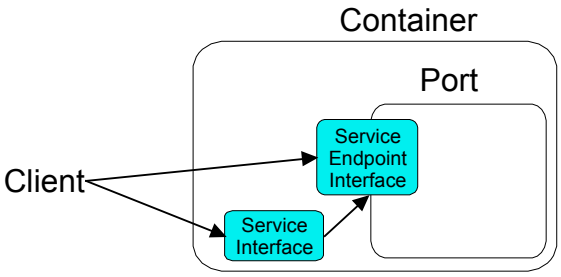
3.2 Web Service

There is no commonly accepted definition for a *Web service*. For the purposes of this specification, a Web service is defined as a component with the following characteristics:

- A service implementation implements the methods of an interface that is describable by WSDL. The methods are implemented using a Stateless Session EJB or JAX-RPC web component.

- A Web service may have its interface published in one or more registries for Web services during deployment.
- A Web Service implementation, which uses only the functionality described by this specification, can be deployed in any Web Services for J2EE compliant application server.
- A service instance, called a Port, is created and managed by a container.
- Run-time service requirements, such as security attributes, are separate from the service implementation. Tools can define these requirements during assembly or deployment.
- A container mediates access to the service.

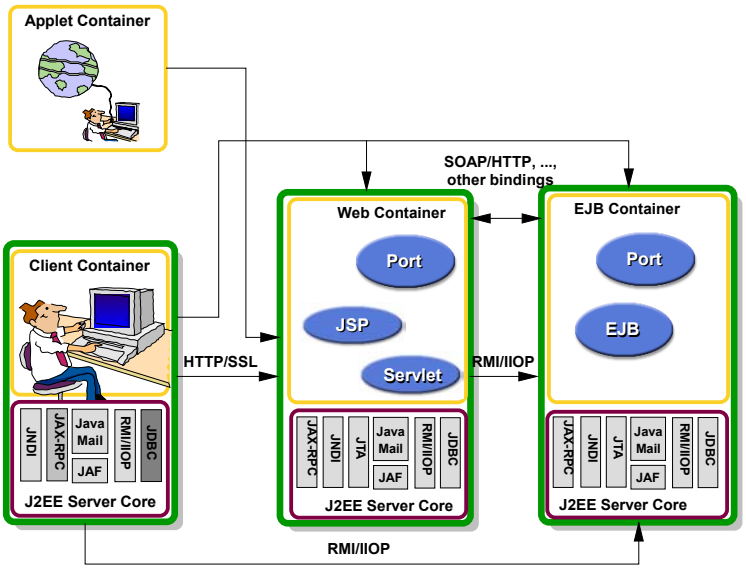
JAX-RPC defines a programming model mapping of a WSDL document to Java which provides a factory (Service) for selecting which aggregated Port a client wishes to use. See Figure 2 for a logical diagram. In general, the transport, encoding, and address of the Port are transparent to the client. The client only needs to make method calls on the Service Endpoint Interface, as defined by JAX-RPC, (i.e. PortType) to access the service. See Chapter 4 for more details.



• Figure 2 Client view

3.3 Web Services for J2EE Overview

The Web Services for J2EE specification defines the required architectural relationships as shown in Figure 3. This is a logical relationship and does not impose any requirements on a container provider for structuring containers and processes. The additions to the J2EE platform include a port component that depends on container functionality provided by the web and EJB containers, and the SOAP/HTTP transport.



• Figure 3 J2EE architecture diagram

Web Services for J2EE requires that a Port be referencable from the client, web, and EJB containers. This specification does not require that a Port be accessible from the applet container.

This specification adds additional artifacts to those defined by JAX-RPC that may be used to implement Web services, a role based development methodology, portable packaging and J2EE container services to the Web services architecture. These are described in later sections.

3.3.1 Web Service Components

This specification defines two means for implementing a Web service, which runs in a J2EE environment, but does not restrict Web service implementations to just those means. The first is a container based extension of the JAX-RPC programming model which defines a Web service as a Java class running in the web container. The second uses a constrained implementation of a stateless session EJB in the EJB container. Other service implementations are possible, but are not defined by this specification.

3.3.2 Web Service Containers

The container provides for life cycle management of the service implementation, concurrency management of method invocations, and security services. A container provides the services specific to supporting Web services in a J2EE environment. This specification does not require that a new container be implemented. Existing J2EE containers may be used and indeed are expected to be used to host Web services. Web service instance life cycle and concurrency management is dependent on which container the service implementation runs in. A JAX-RPC Service Endpoint implementation in a web container follows standard servlet life cycle and concurrency requirements and an EJB implementation in an EJB container follows standard EJB life cycle and concurrency requirements.

3.4 Platform Roles

This specification defines the responsibilities of the existing J2EE platform roles. There are no new roles defined by this specification. There are two roles specific to Web Services for J2EE used within this specification, but they can be mapped onto existing J2EE platform roles. The Web Services for J2EE product provider role can be mapped to a J2EE product provider role and the Web services container provider role can be mapped to a container provider role within the J2EE specification.

In general, the developer role is responsible for the service definition, implementation, and packaging within a J2EE module. The assembler role is responsible for assembling the module into an application, and the deployer role is responsible for publishing the deployed services and resolving client references to services. More details on role responsibilities can be found in later sections.

3.5 Portability

A standard packaging format, declarative deployment model, and standard run-time services provide portability of applications developed using Web services. A Web services specific deployment descriptor included in a standard J2EE module defines the Web service use of that module. More details on Web services deployment descriptors can be found in later chapters. Deployment tools supporting Web Services for J2EE are required to be able to deploy applications packaged according to this specification.

Web services container providers may provide support for additional service implementations and additional transport and encoding bindings at the possible expense of application portability.

3.6 Standard Services

The J2EE platform defines a set of standard services a J2EE provider must supply. The Web Services for J2EE specification identifies an additional set of run-time services which are required.

3.6.1 JAX-RPC

JAX-RPC provides run-time services for marshalling and demarshalling Java data and objects to and from XML SOAP messages. In addition, JAX-RPC defines the WSDL to Java mappings for a Service Endpoint Interface and a Service class.

3.7 Interoperability

This specification extends the interoperability requirements of the J2EE platform by defining interoperability requirements for products that implement this specification on top of J2EE™. The interoperability requirements rely on the interoperability of existing standards that this specification depends on.

The specification builds on the evolving work of the following JSRs and specifications:

- JSR 101, Java™ API for XML-based RPC (JAX-RPC)
- Java 2 Platform Enterprise Edition Specification
- Enterprise JavaBeans Specification
- Java Servlet Specification

3.8 Scope

The following sections define the scope of what is and what is not covered by this specification.

3.8.1 Scope

- The scope of this specification is limited to Web service standards that are widely documented and accepted in the industry. These include:
    - SOAP 1.1 and SOAP with Attachments

- WSDL 1.1
- UDDI 1.0

- This specification is limited to defining support for SOAP over HTTP 1.1 or HTTPS protocols and communication APIs for Web services (vendors are free to support additional transports).
- These standards are expected to continue to change and evolve. Future versions of this JSR will accommodate and address future versions of these standards. In this specification, all references to SOAP, WSDL, and UDDI are assumed to be the versions defined above.

3.8.2 Not in Scope

- The most glaring deficiency of SOAP over HTTP is basic reliable message semantics. Despite this deficiency, this JSR does not consider Message Reliability or Message Integrity to be in scope. Other JSRs, like the evolution and convergence of JAX-M and JMS, as well as activities in W3C and other standard bodies will define these capabilities.
- Persistence of XML data.
- Workflow and data flow models.
- Arbitrary XML transformation.
- Client programming model for Web service clients that do not conform to this specification.

3.9 Web Service Client View

The client view of a Web service is quite similar to the client view of an Enterprise JavaBean. A client of a Web service can be another Web service, a J2EE component, including a J2EE application client, or an arbitrary Java application. A non-Java application or non-Web Services for J2EE application can also be a client of Web service, but the client view for such applications is out of scope of this specification.

The Web service client view is remotable and provides local-remote transparency.

The Port provider and container together provide the client view of a Web service. This includes the following:

- Service interface
- Service Endpoint interface

The JAX-RPC Handler interface is considered a container SPI and is therefore not part of the client view.

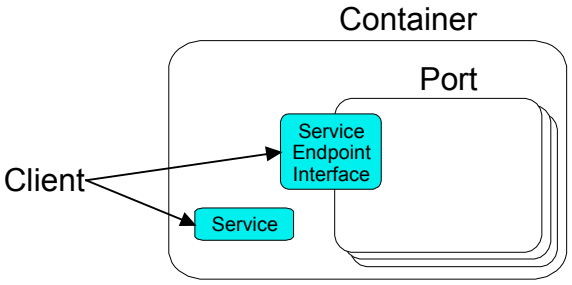


Figure 4 Web Service Client View

The Service Interface defines the methods a client may use to access a Port of a Web service. A client does not create or remove a Port. It uses the Service Interface to obtain access to a Port. The Service interface is defined by the JAX-RPC specification, but its behavior is defined by a WSDL document supplied by the Web

service provider. The container's deployment tools provide an implementation of the methods of the Service Interface or the JAX-RPC Generated Service Interface.

A client locates a Service Interface by using JNDI APIs. This is explained further in Chapter 4.

A Web service implementation is accessed by the client using the Service Endpoint Interface. The Service Endpoint Interface is specified by the service provider. The deployment tools and container run-time provide server side classes which dispatch a SOAP request to a Web service implementation which implements the methods of the Service Endpoint Interface. The Service Endpoint Interface extends the `java.rmi.Remote` interface and is fully defined by the JAX-RPC specification.

A Port has no identity within the client view and is considered a stateless object.

3.10 Web Service Server View

Chapter 5 Server Programming Model defines the details of the server programming model. This section defines the general requirements for the service provider.

The service provider defines the WSDL PortType, WSDL binding, and Service Endpoint Interface of a Web service. The PortType and Service Endpoint Interface must follow the JAX-RPC rules for WSDL->Java and Java->WSDL mapping.

The service provider defines the WSDL service and aggregation of ports in the WSDL document.

The business logic of a Web service is implemented by a service provider in one of two different ways:

1. A Stateless SessionBean: The service provider implements the Web service business logic by implementing a stateless session EJB. The EJB's remote interface method signatures must match the method signatures of the Service Endpoint Interface and must include all methods of the Service Endpoint Interface.
2. A Java class: The service provider implements the Web service business logic according to the requirements defined by the JAX-RPC Servlet based service implementation model.

The life cycle management of a Web service is specific to the service implementation methodology.

The service provider implements the container callback methods specific to the service implementation methodology used. See the JAX-RPC specification and Enterprise JavaBeans specification for details on the container callback methods.

The container manages the run-time services required by the Web service, such as security. A Web service does not execute under a global transaction context. If the client accesses a Port with a transaction context, it will be suspended before the Port is accessed.

Service providers must avoid programming practices that interfere with container operation. These restrictions are defined by the J2EE, Servlet, and EJB specifications.

Packaging of a Web service in a J2EE module is specific to the service implementation methodology, but follows the J2EE requirements for an EJB-JAR file or WAR file. It contains the Java class files of the Service Endpoint Interface and WSDL documents for the Web service. In addition it contains an XML deployment descriptor which defines the Web service Ports and their structure. Packaging requirements are described in Section 5.4 Packaging.

4 Client Programming Model

This chapter defines the client programming model of Web Services for J2EE. In general, the client programming model is covered in detail by the JAX-RPC specification. This specification covers the use of the JAX-RPC client programming model in a J2EE environment.

Differences between this specification and the JAX-RPC specification will be noted in this style.

4.1 Concepts

Clients of Web services are not limited to clients defined within this specification, however the client programming model for non-Web Services for J2EE clients is not specifically addressed by this specification. In general, the WSDL definition of a Web service provides enough information for a non-Web Services for J2EE client to be built and run, but the programming model for that is undefined. The rest of this chapter covers the programming model for Web Services for J2EE clients. It makes no assumption on whether the Web service implementation invoked by the client is hosted by a Web Services for J2EE run-time or some external run-time.

A client uses the Web Services for J2EE run-time to access and invoke the methods of a Web service. A client can be any of the following: J2EE application client, web component, EJB component, or another Web service.

The client view of a Web service is a set of methods that perform business logic on behalf of the client. A client cannot distinguish whether the methods are being performed locally or remotely, nor can the client distinguish how the service is implemented. Lastly, a client must assume that the methods of a Web service have no state that is persistent across multiple Web service method invocations. A client can treat the Web service implementation as stateless.

A client accesses a Web service using a Service Endpoint Interface as defined by the JAX-RPC specification. 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.

A client uses JNDI lookup to access a Service object that implements the Service Interface as defined by the JAX-RPC specification. The Service object is a factory used by the client to get a stub or proxy that implements the Service Endpoint Interface. The stub is the client representation of an instance of the Web service.

The Service Interface can be a generic `javax.xml.rpc.Service` interface or a Generated Service Interface, which extends `javax.xml.rpc.Service`, as defined by JAX-RPC. Further references in this document to the Service Interface refer to either the generic or generated version, unless noted otherwise.

The client has no control over the life cycle of the Web service implementation on the server. A client does not create or destroy instances of a Web service, which is referred to as a Port. The client only accesses the Port. The life cycle of the Ports, or instances of a Web service implementation, are managed by the run-time that hosts the Web service. A Port has no identity. This means that a client cannot compare a Port to other Ports to see if they are the same or identical, nor can a client access a specific Port instance. A client cannot tell if a server crashes and restarts if the crash and restart complete in between Web service access.

A client developer starts with the Service Endpoint Interface and Service Interface. How a developer obtains these is out of scope, but includes having the Web service provider supply them or tools generate them from a WSDL definition supplied by the Web service provider. These tools operate according to the JAX-RPC rules

for WSDL->Java mapping. A client developer does not need to generate stubs during development, nor are they encouraged to do so. The client should use the interfaces, and not the stubs. Stubs will be generated during deployment and will be specific to the vendor's run-time the client will run in.

Each client JNDI lookup of a Web service is by a logical name. A client developer chooses the logical name to be used in the client code and declares it along with the required Service Interface in a Web service client deployment descriptor. The client should use the interfaces, and not the stubs.

The Service Interface methods can be categorized into two groups: stub/proxy and DII. The stub/proxy methods provide both service specific (client requires WSDL knowledge) and service agnostic (does not require WSDL knowledge) access to Ports. The DII methods are used when a client needs dynamic, non-stub based communication with the Web service.

A client can use the stub/proxy methods of the Service Interface to get a Port stub or dynamic proxy. The WSDL specific methods can be used when the full WSDL definition of the service is available to the client developer. The WSDL agnostic methods must be used if the client developer has a partial WSDL definition that only contains only the portType and bindings.

4.2 Specification

The following sections define the requirements for J2EE product providers that implement Web Services for J2EE and developers for creating applications that run in such an environment.

4.2.1 Service Lookup

The client developer is required to define a logical JNDI name for the Web service called a service reference. This name is specified in the deployment descriptor for the client. It is recommended, but not required that all service reference logical names be organized under the `service` subcontext of a JNDI name space. The container must bind the Service Interface implementation under the client's environment context, `java:comp/env,` using the logical name of the service reference. In the following examples, the logical service name declared in the client deployment descriptor is `service/AddressBookService`.

The container acts as a mediator on behalf of the client to ensure a Service Interface is available via a JNDI lookup. More specifically, the container must ensure that an implementation of the required Service Interface is bound at a location in the JNDI namespace of the client's choosing as declared by the service reference in the Web services client deployment descriptor. This is better illustrated in the following code segment:

```
InitialContext ic = new InitialContext ();
Service abf = (Service)ic.lookup(
       "java:comp/env/service/AddressBookService");
```

In the above example, the container must ensure that an implementation of the generic Service Interface, `javax.xml.rpc.Service`, is bound in the JNDI name space at a location specified by the developer. A similar code fragment is used for access to an object that implements a Generated Service Interface such as AddressBookService.

```
InitialContext ic = new InitialContext ();
AddressBookService abf = (AddressBookService)ic.lookup(
       "java:comp/env/service/AddressBookService");
```

A J2EE product provider is required to provide Service lookup support in the web, EJB, and application client containers.

4.2.2 Service Interface

The Service Interface is used by a client to get a stub or dynamic proxy or a DII Call object for a Port. A container provider is required to support all methods of the Service interface except for the getHandlerRegistry() and getTypeMappingRegistry() methods as described in sections 4.2.2.8 and 4.2.2.9.

A client developer must declare the Service Interface type used by the application in the client deployment descriptor. The Service Interface represents the deployed WSDL of a service.

4.2.2.1 Stub/proxy access

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

```
java.rmi.Remote getPort(QName portName, Class serviceEndpointInterface)
    throws ServiceException;
java.rmi.Remote getPort(java.lang.Class serviceEndpointInterface)
    throws ServiceException;
```

The client may also use the additional methods of the Generated Service Interface to obtain a static stub or dynamic proxy for a Web service.

The container must provide at least one of static stub or dynamic proxy support for these methods as described in section 4.2.3. The container must ensure the stub or dynamic proxy is fully configured for use by the client, before it is returned to the client. The deployment time choice of whether a stub or dynamic proxy is returned by the `getPort` or `get<port name>` methods is out of the scope of this specification. Container providers are free to offer either one or both.

The container provider must provide Port resolution for the `getPort(java.lang.Class serviceEndpointInterface)` method. This is useful for resolving multiple WSDL ports that use the same binding or when ports are unknown at development time. A client must declare its dependency on container Port resolution for a Service Endpoint Interface in the client deployment descriptor. If a dependency for resolving the interface argument to a port is not declared in the client deployment descriptor, the container may provide a default resolution capability or throw a ServiceException.

4.2.2.2 Dynamic Port access

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() throws ServiceException;
Call createCall(QName portName) throws ServiceException;
Call createCall(QName portName, String operationName) throws
    ServiceException;
Call createCall(QName portName, QName operationName) throws
    ServiceException;
Call[] getCalls(QName portName) throws ServiceException;
```

A DII Call object may or may not be pre-configured for use depending on the method used to obtain it. See the JAX-RPC specification for details.

4.2.2.3 ServiceFactory

Use of the JAX-RPC `ServiceFactory` class is not recommended in a Web Services for J2EE product. A Web Services for J2EE client must obtain a Service Interface using JNDI lookup as described in section 4.2.1. Container providers are not required to support managed Service instances created from a ServiceFactory.

4.2.2.4 Service method use with full WSDL

A client developer may use all methods of the Service Interface (except as described in sections 4.2.2.8 and 4.2.2.9) if a full WSDL description and JAX-RPC mapping file are declared in the client deployment descriptor. The port address attribute may be absent from the WSDL or may be a dummy value.

If a client developer uses the getPort(SEI) method of a Service Interface and the WSDL supports multiple ports the SEI could be bound to, the developer can indicate to a deployer a binding order preference by ordering the ports in the service-ref's WSDL document.

4.2.2.5 Service method use with partial WSDL

A client developer may use the following methods of the Service Interface if a partial WSDL definition is declared in the client deployment descriptor:

```
Call createCall() throws ServiceException;
java.rmi.Remote getPort(java.lang.Class serviceEndpointInterface) throws
    ServiceException;
javax.xml.namespace.QName getServiceName();
java.util.Iterator getPorts() throws ServiceException;
java.net.URL getWSDLDocumentLocation()
```

A partial WSDL definition is defined as a fully specified WSDL document which contains no service or port elements. The JAX-RPC mapping file specified by the developer will not include a `service-interface-mapping` in this case.

Use of other methods of the Service Interface is not recommended when a developer specifies a partial WSDL definition. The behavior of the other methods is unspecified.

The container must provide access to all SEIs declared by the `port-component-ref` elements of the `service-ref` through the getPort(SEI) method.

4.2.2.6 Service method use with no WSDL

A client developer may use the following methods of the Service Interface if no WSDL definition is specified in the client deployment descriptor:

```
Call createCall() throws ServiceException;
```

If the `wsdl-file` is not specified in the deployment descriptor, the `jaxrpc-mapping-file` must not be specified.

Use of other methods of the Service Interface is not recommended. Their behavior is unspecified.

4.2.2.7 Service Interface method behavior

The following table summarizes the behavior of the methods of the Service Interface under various deployment configurations.

• Table 1 Service Interface method behavior

Method	Full WSDL	Partial WSDL	No WSDL
Call createCall()	Normal	Normal	Normal
Call createCall(QName port)	Normal	Unspecified	Unspecified
Call createCall(QName port, QName operation)	Normal	Unspecified	Unspecified
Call createCall(QName port, String operation)	Normal	Unspecified	Unspecified
Call[] getCalls(QName port)	Normal	Unspecified	Unspecified
HandlerRegistry getHandlerRegistry()	Exception[1]	Exception[1]	Exception[1]
Remote getPort(Class SEI)	Normal	Normal	Unspecified
Remote getPort(QName port, Class SEI)	Normal	Unspecified	Unspecified
Iterator getPorts()	Bound ports	Bound ports	Unspecified
QName getServiceName()	Bound service name	Bound service name	Unspecified
TypeMappingRegistry getTypeMappingRegistry()	Exception[1]	Exception[1]	Exception[1]
URL getWSDLDocumentLocation()	Bound WSDL location	Bound WSDL location	Unspecified

[1]See sections 4.2.2.8 and 4.2.2.9.

4.2.2.8 Handlers

Components should not use the `getHandlerRegistry()` method. A container provider must throw a `java.lang.UnsupportedOperationException` from the `getHandlerRegistry()` method of the Service Interface. Handler support is documented in Chapter 6 Handlers.

4.2.2.9 Type Mapping

Components should not use the `getTypeMappingRegistry()` method. A container provider must throw a java.lang.UnsupportedOperationException from the `getTypeMappingRegistry()` method of the Service Interface.

4.2.3 Port Stub and Dynamic Proxy

The following sections define the requirements for implementing and using static Stubs and Dynamic Proxies.

4.2.3.1 Identity

The Port Stub and Dynamic Proxy are a client's representation of a Web service. The Port that a stub or proxy communicates with has no identity within the client view. The equals() method cannot be used to compare two stubs or proxy instances to determine if they represent the same Port. The results of the `equals(), hash(), and toString()` methods for a stub are unspecified. There is no way for the client to ensure that a Port Stub, Dynamic Proxy, or Call will access a particular Port instance or the same Port instance for multiple invocations.

4.2.3.2 Type narrowing

Although the stub and dynamic proxy classes are considered Remote objects, a client is not required to use `PortableRemoteObject.narrow`(…). However, clients are encouraged to use `PortableRemoteObject.narrow`(…) to prevent confusion with client use of other Remote objects.

4.2.4 JAX_RPC Properties

The J2EE container environment provides a broader set of operational characteristics and constraints for supporting the Stub/proxy properties defined within JAX-RPC. While support of the JAX-RPC required standard properties for Stub and Call objects is required, their use may not work in all cases in a J2EE environment.

The following properties are not recommended for use in a managed context defined by this specification:

- `javax.xml.rpc.security.auth.username`
- `javax.xml.rpc.security.auth.password`

4.2.4.1 Required properties

A container provider is required to support the `javax.xml.rpc.service.endpoint.address` property to allow components to dynamically redirect a Stub/proxy to a different URI.

4.2.5 JAX-RPC Custom Serializers / Deserializers

The use of JAX-RPC custom serializers / deserializers is out of scope for this version of the specification. JAX-RPC customer serializers / deserializers are not portable across Web Services for J2EE providers and are therefore not included as part of the portable deployment unit. It is expected that vendors will provide proprietary solutions to this problem until it has been addressed by a future version of JAX-RPC.

4.2.6 Packaging

The developer is responsible for packaging, either by containment or reference (i.e. by using the MANIFEST ClassPath to refer to other JAR files that contain the required classes), the class files for each Web service including the: Service Endpoint Interface classes, Generated Service Interface class (if used), and their dependent classes. The following files must also be packaged in the module: WSDL files, JAX-RPC Mapping files, and a Web services client deployment descriptor in a J2EE module. The location of the Web services client deployment descriptor in the module is module specific. WSDL files are located relative to the root of the module and are typically co-located with the module's deployment descriptor. JAX-RPC Mapping Files are located relative to the root of the module and are typically co-located with the WSDL file. The developer must not package generated stubs.

5 Server Programming Model

This chapter defines the server programming model for Web Services for J2EE. A WSDL document defines the interoperability of Web services and includes the specification of transport and wire format requirements. In general, WSDL places no requirement on the programming model of the client or the server. Web Services for J2EE defines two methods of implementing a Web service. It requires the JAX-RPC Servlet container based Java class programming model for implementing Web services that run in the web container and it requires the Stateless Session EJB programming model for implementing Web services that run in the EJB container. These two implementation methods provide a means for defining a Port component to bring portable applications into the Web Services programming paradigm. This specification also requires that a developer be able to start simple and grow up to use more complex qualities of service. The following sections define the requirements for Port components.

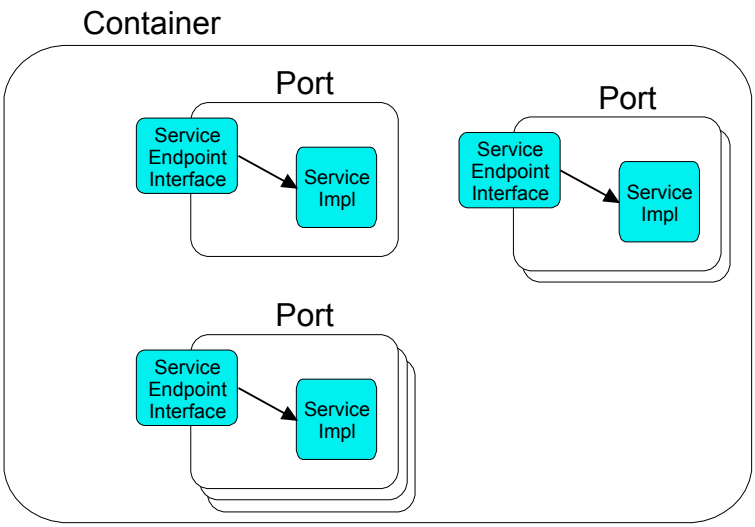
5.1 Goals

Port components address the following goals:

- Provide a portable Web services programming model
- Provide a server programming model which maintains a consistent client view. The client must not be required to know how the service is implemented.
- Provide path to start simple and grow to more complex run-time service requirements
- Leverage existing J2EE container functionality
- Leverage familiar programming models

5.2 Concepts

A Port component (sometimes referred to as Port) defines the server view of a Web service. Each Port services a location defined by the WSDL port address. A Port component services the operation requests defined by a WSDL PortType. Every Port component has a Service Endpoint Interface and a Service Implementation Bean. The Service Endpoint Interface is a Java mapping of the WSDL PortType and binding associated with a WSDL port. The Service Implementation Bean can vary based on the container the Port is deployed in, but in general it is a Java class which implements the methods defined by the Service Endpoint Interface. WSDL ports, which differ only in address, are mapped to separate Port components, each with its own potentially unique but probably shared Service Implementation Bean. Figure 5 illustrates this below.



• Figure 5 container

A Port's life cycle is specific to and completely controlled by the container, but in general follows the same life cycle of the container itself. A Port is created and initialized by the container before the first request received at the WSDL port address can be serviced. A Port is destroyed by the container whenever the container feels it is necessary to do so, such as when the container is shutting down.

The implementation of a Port and the container it runs in are tied. A JAX-RPC Service Implementation Bean always runs in a web container. An EJB Service Implementation Bean always runs in an EJB container.

The Port component associates a WSDL port address with a Service Implementation Bean. In general the Port component defers container service requirement definition to the J2EE component's deployment descriptor. This is discussed further in Chapters 6.3 and 7.3. A container provides a listener for the WSDL port address and a means of dispatching the request to the Service Implementation. A container also provides run-time services such as security constraints and logical to physical mappings for references to distributed objects and resources.

5.3 Port Component Model Specification

A Port component defines the programming model artifacts that make the Web Service a portable server application. The association of a Port component with a WSDL port provides for interoperability. The programming model artifacts include:

WSDL document – Although not strictly a programming model artifact, the WSDL document provides a canonical description of a Web service that may be published to third parties. A WSDL document and the Service Endpoint Interface are related by the JAX-RPC WSDL<->Java mapping rules. Support for OUT and IN/OUT parameters is optional for the EJB container.

Service Endpoint Interface (SEI) - This interface defines the methods that are implemented by the Service Implementation Bean. Support for Holder parameters is optional for the EJB container.

Service Implementation Bean - The Service Implementation Bean is a Java class that provides the business logic of the Web service. In addition, it defines the Port component contract for the container, which allows the business logic to interact with container services. It implements the same

methods and signatures of the SEI, but is not required to implement the SEI itself.

Security Role References - The Port may declare logical role names in the deployment descriptor. These logical role names are reconciled across the modules by the assembler and mapped to physical roles at deployment time and allow the service to provide instance level security checks.

A developer declares a Port component within a Web services deployment descriptor. The deployment descriptor includes the WSDL document that describes the PortType and binding of the Web service. A deployer and the deploy tool handles the mapping of the Port into a container.

5.3.1 Service Endpoint Interface

The Service Endpoint Interface (SEI) must follow the JAX-RPC rules for WSDL<->Java mapping. The SEI is related to the WSDL PortType and WSDL bindings by these rules. The SEI is required for use by the deployment tools and parallel client development. The Port component developer is responsible for providing both the WSDL document with a minimum of the PortType and binding defined and the SEI and for keeping the two in sync with each other.

JAX-RPC defines Holders as non-serializable classes which cannot be implemented by the remote interface of an Enterprise JavaBean. Therefore, support for an SEI which uses Holders for parameters is not required for Port components deployed in the EJB container.

5.3.2 Service Implementation Bean

There are two ways a Service Implementation Bean can be implemented. This includes a Stateless Session EJB and JAX-RPC service endpoint as defined by Chapter 10 of the JAX-RPC specification. The two programming models are fully defined in sections 5.3.2.1 and 5.3.2.2.

A container may use any bean instance to service a request.

5.3.2.1 EJB container programming model

A Stateless Session Bean, as defined by the Enterprise JavaBeans specification, can be used to implement a Web service to be deployed in the EJB container.

A Stateless Session Bean does not have to worry about multi-threaded access. The EJB container is required to serialize request flow through any particular instance of a Service Implementation Bean.

The requirements for creating a Service Implementation Bean as a Stateless Session EJB are repeated in part here.

- The Service Implementation Bean must have a default public constructor.
- The Service Implementation Bean may implement the Service Endpoint Interface, but it is not required to do so. The bean must implement all the method signatures of the SEI. The Service Implementation Bean methods are not required to throw javax.rmi.RemoteException. The business methods of the bean must be public and must not be final or static. It may implement other methods in addition to those defined by the SEI.
- A Service Implementation Bean must be a stateless object. A Service Implementation Bean must not save client specific state across method calls either within the bean instance's data members or external to the instance.
- The class must be public, must not be final and must not be abstract.

- The class must not define the `finalize()` method.
- Currently, it must implement the `ejbCreate()` and `ejbRemove()` methods which take no arguments. This is a requirement of the EJB container, but generally can be stubbed out with an empty implementations.

5.3.2.1.1 The required SessionBean interface

Currently, a Stateless Session Bean must implement the `javax.ejb.SessionBean` interface either directly or indirectly.

This interface allows the container to notify the Service Implementation Bean of impending changes in its state. The full requirements of this interface are defined in the Enterprise JavaBeans specification section 7.5.1.

5.3.2.1.2 Allowed access to container services

The Enterprise JavaBeans specification section 7.8.2 defines the allowed container service access requirements.

5.3.2.1.3 Exposing an existing EJB

An existing Enterprise JavaBean may be used as a Service Implementation Bean if it meets the following requirements:

- The business methods of the EJB bean class that are exposed on the SEI must meet the Service Implementation Bean requirements defined in section 5.3.1.
- The Service Endpoint Interface methods must be a subset of the remote interface methods of the EJB and the SEI must meet the requirements described in the JAX-RPC specification for Java->WSDL mapping.
- The transaction attributes of the SEI methods must not include Mandatory.

The developer must package the Web service as described in section 5.4 and must specify an `ejb-link` from the port in the Web services deployment descriptor to the existing EJB.

5.3.2.2 Web container programming model

The term JAX-RPC Service Endpoint used within the JAX-RPC specification is somewhat confusing since both Service Implementation Beans require the use of a JAX-RPC run time. However, in this case it refers to the programming model defined within the JAX-RPC specification that is used to create Web services that run within the web container. The requirements are repeated here with clarification. Changes from the JAX-RPC defined programming model are required for running in a J2EE container-managed environment.

A JAX-RPC Service Endpoint can be single or multi-threaded. The concurrency requirement is declared as part of the programming model. A JAX-RPC Service Endpoint must implement `javax.servlet.SingleThreadModel` if single threaded access is required by the component. A container must serialize method requests for a Service Implementation Bean that implements the `SingleThreadModel` interface.

The Service Implementation Bean must follow the Service Developer requirements outlined in the JAX-RPC specification and are listed below except as noted.

- The Service Implementation Bean must have a default public constructor.
- The Service Implementation Bean may implement the Service Endpoint Interface as defined by the JAX-RPC Servlet model. The bean must implement all the method signatures of the SEI. In addition, a Service Implementation Bean may be implemented that does not implement the SEI. This additional requirement

provides the same SEI implementation flexibility as provided by EJB service endpoints. The business methods of the bean must be public and must not be static. If the Service Implementation Bean does not implement the SEI, the business methods must not be final. The Service Implementation Bean may implement other methods in addition to those defined by the SEI, but only the SEI methods are exposed to the client.

- A Service Implementation must be a stateless object. A Service Implementation Bean must not save client specific state across method calls either within the bean instance's data members or external to the instance. A container may use any bean instance to service a request.

- The class must be public, must not be final and must not be abstract.
- The class must not define the finalize() method.

5.3.2.2.1 The optional ServiceLifecycle Interface

A Service Implementation Bean for the web container may implement the `java.xml.rpc.server.ServiceLifeCycle` interface:

```
package javax.xml.rpc.server;
public interface ServiceLifecycle {
       void init(Object context) throws ServiceException;
       void destroy();
}
```

The `ServiceLifeCycle` interface allows the web container to notify a Service Implementation Bean instance of impending changes in its state. The bean may use the notification to prepare its internal state for the transition. If the bean implements the `ServiceLifeCycle` interface, the container is required to call the `init` and `destroy` methods as described below.

The container must call the `init` method before it can start dispatching requests to the SEI methods of the bean. The `init` method parameter value provided by the container is described by the JAX-RPC specification. The bean may use the container notification to ready its internal state for receiving requests.

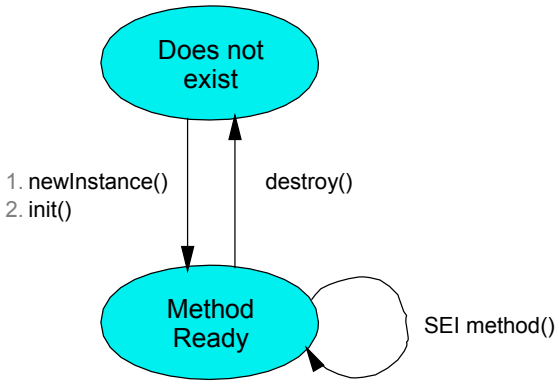
The container must notify the bean of its intent to remove the bean instance from the container's working set by calling the `destroy` method. A container may not call the `destroy` method while a request is being processed by the bean instance. The container may not dispatch additional requests to the SEI methods of the bean after the `destroy` method is called.

5.3.2.2.2 Allowed access to container services

The container provides certain services based on the life cycle state of the Service Implementation Bean. Access to services provided by a web container in a J2EE environment (e.g. transactions, JNDI access to the component's environment, etc.) must follow the requirements defined by the Servlet and J2EE specifications. Access to a `ServletEndpointContext` must meet the requirements defined by the JAX-RPC specification section 10.1.3.

5.3.3 Service Implementation Bean Life Cycle

The life cycle of a Service Implementation Bean is controlled by the container and is illustrated in Figure 6. The methods called by the container are container/bean specific, but in general are quite similar. Figure 6 illustrates the life cycle in the web container. The EJB container life cycle may be found in the Enterprise JavaBeans specification section 7.8.1.



• Figure 6 Service Implementation Bean life cycle in the web container

The container services requests defined by a WSDL port. It does this by creating a listener for the WSDL port address, receiving requests and dispatching them on a Service Implementation Bean. Before a request can be serviced, the container must instantiate a Service Implementation Bean and ready it for method requests.

A container readies a bean instance by first calling `newInstance` on the Service Implementation Bean class to create an instance. The container then calls the life cycle methods on the Service Implementation Bean that are specific to the container. For the web container, it calls the `init` method on the instance if the Service Implementation Bean class implements the `ServiceLifecycle` interface. For the EJB container, it calls the `setSessionContext` and `ejbCreate` methods.

A Service Implementation Bean instance has no identity.

A container may pool method ready instances of a Service Implementation Bean and dispatch a method request on any instance in a method ready state.

The container notifies a Service Implementation Bean instance that it is about to be removed from Method Ready state by calling container/bean specific life cycle methods on the instance. For the web container, the `destroy` method is called. For the EJB container, the `ejbRemove` method is called.

5.3.4 JAX-RPC Custom Serializers / Deserializers

The use of JAX-RPC custom serializers / deserializers is out of scope for this version of the specification. JAX-RPC customer serializers / deserializers are not portable across Web Services for J2EE providers and are therefore not included as part of the portable deployment unit. It is expected that vendors will provide proprietary solutions to this problem until it has been addressed by a future version of JAX-RPC.

5.4 Packaging

Port components may be packaged in a WAR file, or EJB JAR file. Port components packaged in a WAR file must use a JAX-RPC Service Endpoint for the Service Implementation Bean. Port components packaged in a EJB-JAR file must use a Stateless Session Bean for the Service Implementation Bean.

The developer is responsible for packaging, either by containment or reference, the WSDL file, Service Endpoint Interface class, Service Implementation Bean class, and their dependent classes, JAX-RPC mapping file along with a Web services deployment descriptor in a J2EE module. The location of the Web services deployment descriptor in the module is module specific. WSDL files are located relative to the root of the module and are typically co-located with the module deployment descriptor. Mapping files are located relative to the root of the module and are typically co-located with the WSDL file.

5.4.1 EJB Module Packaging

Stateless Session EJB Service Implementation Beans are packaged in an EJB-JAR that contains the class files and WSDL files. The packaging rules follow those defined by the Enterprise JavaBeans specification. In addition, the Web services deployment descriptor location within the EJB-JAR file is `META-INF/webservices.xml`.

5.4.2 Web App Module Packaging

JAX-RPC Service Endpoints are packaged in a WAR file that contains the class files and WSDL files. The packaging rules for the WAR file are those defined by the Servlet specification. A Web services deployment descriptor is located in a WAR at WEB-INF/webservices.xml.

5.4.3 Assembly within an EAR file

Assembly of modules containing port components into an EAR file follows the requirements defined by the J2EE specification.

5.5 Transactions

The methods of a Service Implementation Bean run under a transaction context specific to the container. The web container runs the methods under an unspecified transaction context. The EJB container runs the methods under the transaction context defined by the `container-transaction` element of the EJB deployment descriptor.

5.6 Container Provider Responsibilities

In addition to the container requirements described above a container provider must provide a JAX-RPC runtime.

It is the responsibility of the container provider to support processing JAX-RPC compliant requests and invoking Ports as described above. The application server must support deployment of these Ports. This specification prescribes the use of the JAX-RPC Java<->WSDL and Java<->XML Serialization framework for all XML Protocol based Web service bindings. For JAX-RPC inbound messages, the container will act as the JAX-RPC server side runtime. It is responsible for:

1. Listening on a well known port or on the URI of the Web service implementation (as defined in the service's WSDL after deployment) for SOAP/HTTP bindings.
2. Parsing the inbound message according to the Service binding.
3. Mapping the message to the implementation class and method according to the Service deployment data.
4. Creating the appropriate Java objects from the SOAP envelope according to the JAX-RPC specification.
5. Invoking the Service Implementation Bean handlers and instance method with the appropriate Java parameters.
6. Capturing the response to the invocation if the style is request-response
7. Mapping the Java response objects into SOAP message according to the JAX-RPC specification.
8. Creating the message envelope appropriate for the transport

9. Sending the message to the originating Web service client.

6 Handlers

This chapter defines the programming model for handlers in Web Services for J2EE. Handlers define a means for an application to access the raw SOAP message of a request. This access is provided on both the client and server. Handlers are not part of the WSDL specification and are therefore not described in it. See chapter 6.3 for declaration of handlers within deployment descriptors. The JAX-RPC specification defines the Handler APIs in chapter 12. This specification defines Handler use within a J2EE environment.

6.1 Concepts

A Handler can be likened to a Servlet Filter in that it is business logic that can examine and potentially modify a request before it is processed by a Web Service component. It can also examine and potentially modify the response after the component has processed the request. Handlers can also run on the client before the request is sent to the remote host and after the client receives a response.

JAX-RPC Handlers are specific to SOAP requests only and cannot be used for other non-SOAP Web services. Handlers may be transport independent. For instance, a Handler as defined by JAX-RPC may be usable for SOAP/JMS in addition to SOAP/HTTP if a JMS protocol binding was available. Handlers for non-SOAP encodings have not been defined yet.

Handlers are service specific and therefore associated with a particular Port component or port of a Service interface. This association is defined in the deployment descriptors in section 7.1 and 7.2 respectively. They are processed in an ordered fashion called a HandlerChain, which is defined by the deployment descriptors.

There are several scenarios for which Handlers may be considered. These include application specific SOAP header processing, logging, and caching. A limited form of encryption is also possible. For application specific SOAP header processing, it is important to note that the client and server must agree on the header processing semantics without the aid of a WSDL description that declares the semantic requirements. Encryption is limited to a doc/literal binding in which the SOAP message part maps to a SOAPElement. In this case, a value within the SOAPElement may be encrypted as long as the encryption of that value does not change the structure of the SOAPElement.

Some Handler scenarios described within the JAX-RPC specification on not supported by this specification. For example, auditing cannot be fully supported because there is no means for a Handler to obtain the Principal. The secure stock quote example cannot be supported as stated because encrypting the body would prevent the container from determining which Port component the request should be directed to and therefore which Handler should decrypt the body.

A Handler always runs under the execution context of the application logic. On the client side, the Stub/proxy controls Handler execution. 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. Handlers can access the java:comp/env context for accessing resources and environment entries defined by the Port component the Handler is associated with.

Handlers are constrained by the J2EE managed environment. Handlers are not able to re-target a request to a different component. Handlers cannot change the WSDL operation nor can Handlers change the message part types and number of parts. On the server, Handlers can only communicate with the business logic of the component using the MessageContext. On the client, Handlers have no means of communicating with the business logic of the client. 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.

The life cycle of a Handler is controlled by the container.

Handlers are associated with the Port component on the server and therefore run in both the web and EJB containers. This specification makes Handler support in the EJB container optional due to the required EJB container changes that would be necessary to implement Handler support. It is expected that EJB Handler support will be made required in a future J2EE specification.

6.2 Specification

This section defines the requirements for JAX-RPC Handlers running in Web Services for J2EE. Chapter 12 of the JAX-RPC specification defines the programming model requirements. Differences between this specification and the JAX-RPC specification are noted in boxed paragraphs.

6.2.1 Scenarios

Handlers must be able to support the following scenarios:

Scenario 1: Handlers must be able to transform the SOAP header. One example is the addition of a SOAP header for application specific information, like customerId, by the handler.

Scenario 2: Handlers must be able to transform just parts of the body. This might include changing part values within the SOAP body. Encryption of some parameter values is an example of this scenario.

Scenario 3: Handlers must be able to just read a message where no additions, transformations, or modification to the message is made. Common scenarios are logging, metering, and accounting.

6.2.2 Programming Model

A Web Services for J2EE provider is required to provide all interfaces and classes of the `javax.xml.rpc.handler` package.

The `HandlerInfo setHandlerConfig()` and `getHandlerConfig()` methods do not affect the container's Handler request processing.

A Web Services for J2EE provider is not required to provide an implementation of `HandlerRegistry`. This functionality is specific to the container.

A Web Services for J2EE provider is required to provide an implementation of `MessageContext`.

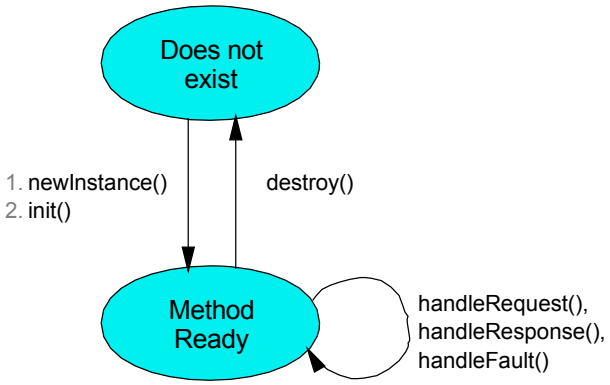
A Web Services for J2EE provider is required to provide all the interfaces of the `javax.xml.rpc.handler.soap` package. The provider must also provide an implementation of the `SOAPMessageContext` interface.

The programming model of a Port component can be single-threaded or multi-threaded as defined in sections 5.3.2.1 and 5.3.2.2. The concurrency of a JAX-RPC Handler must match the concurrency of the business logic it is associated with. Client handlers may need to support multi-threaded execution depending on the business logic which is accessing the Port.

Handlers must be loaded using the same class loader the application code was loaded with. The class loading rules follow the rules defined for the container the Handler is running in.

6.2.2.1 Handler Life Cycle

The life cycle of a Handler is controlled by the container and is illustrated in Figure 7.



• Figure 7 Handler life cycle

The `init` and `destroy methods` of the Handler interface allows the container to notify a Handler instance of impending changes in its state. The Handler may use the notification to prepare its internal state for the transition. The container is required to call the `init` and `destroy` methods as described below.

The container must call the `init` method before it can start dispatching requests to the `handleRequest()`, `handleResponse()`, and `handleFault()` methods of the Handler. The Handler may use the container notification to ready its internal state for receiving requests.

The container must notify the Handler of its intent to remove the instance from the container's working set by calling the `destroy` method. A container must not call the `destroy` method while a request is being processed by the Handler instance. The container must not dispatch additional requests to the Handler interface methods after the `destroy` method is called.

As defined by JAX-RPC, a `RuntimeException` (other than `SOAPFaultException`) thrown from any method of the `Handler` results in the `destroy` method being invoked and transition to the "Does Not Exist" state.

Pooling of Handler instances is allowed, but is not required. If Handler instances are pooled, they must be pooled by Port component. This is because Handlers may retain non-client specific state across method calls that are specific to the Port component. For instance, a Handler may initialize internal data members with Port component specific environment values. These values may not be consistent when a single Handler type is associated with multiple Port components. Any pooled instance of a Port component's Handler in a Method Ready state may be used to service the `handleRequest()`, `handleResponse()`, and `handleFault()` methods. It is not required that the same Handler instance service both the `handleRequest()` and `handleResponse()` or `handleFault()` method invocations of any given request.

6.2.2.2 Security

Handlers associated with a Port component run after authorization has occurred and before the business logic method of the Service Implementation bean is dispatched to. For JAX-RPC Service endpoints, Handlers run after the container has performed the security constraint checks associated with the servlet element that defines the Port component. For EJB based service implementations, Handlers run after method level authorization has occurred.

A Handler must not change the message in any way that would cause the previously executed authorization check to execute differently. A `Handler.handleRequest()` method must not change the operation name, number of parts in the message, or types of the message parts. A container must throw a SOAP fault with a `faultcode` of `soap-env:Server` (the namespace identifier for the namespace prefix, `soap-env:`, is `http://www.w3.org/2001/09/soap-envelope`) back to the client if the Handler does this. Although, not strictly required for security reasons, a `Handler.handleResponse()` method

must not change the number of parts in the message, or types of the message parts. A container must throw a `SOAP fault with a faultcode of soap-env:Server` back to the client if the Handler does this. A container should log occurrences of these errors since the client may not be expecting a response (i.e. it may be a one-way invocation).

A handler may perform programmatic authorization checks if the authorization is based solely on the MessageContext and the component's environment values. A Handler cannot perform role based programmatic authorization checks nor can a Handler access the Principal associated with the request.

The Java 2 security permissions of a Handler follow the permissions defined by the container it runs in. The application client, web, and EJB containers may have different permissions associated with them. If the provider allows defining permissions on a per application basis, permissions granted to a Handler are defined by the permissions granted to the application code it is packaged with. See section J2EE.6.2.3 of the J2EE specification for more details.

6.2.2.3 Transactions

Handlers run under the transaction context of the component they are associated with.

Handlers must not demarcate transactions using the `javax.transaction.UserTransaction` interface.

6.2.3 Developer Responsibilities

A developer is not required to implement a Handler. Handlers are another means of writing business logic associated with processing a Web services request. A developer may implement zero or more Handlers that are associated with a Port component and/or a Service reference. If a developer implements a Handler, they must follow the requirements outlined in this section.

A Handler is implemented as a stateless instance. A Handler does not maintain any message processing (client specific) related state in its instance variables across multiple invocations of the handle method.

A Handler class must implement the java.xml.rpc.handler.Handler interface.

A Handler.handle<action>() method may access the component's environment entries by using JNDI lookup of the `"java:comp/env"` contenxt and accessing the `env-entry-name`s defined in the deployment descriptor by performing a JNDI lookup. See chapter 20 of the Enterprise JavaBeans specification for details. The container may throw a `java.lang.IllegalStateException` if the environment is accessed from any other Handler method and the environment is not available. In addition, the Handler may use HandlerInfo.getHandlerConfig() method to access the Handler's `init-param`s declared in the deployment descriptor.

The Handler.init() method must retain the information defined by HandlerInfo.getHeaders().

A Handler implementation must implement the getHeaders() method to return the results of the HandlerInfo.getHeaders() method. The headers that a Handler declares it will process (i.e. those returned by the Handler.getHeaders() method must be defined in the WSDL definition of the service.

A Handler implementation should test the type of the MessageContext passed to the Handler in the handle<action>() methods. Although this specification only requires support for SOAP messages and the container will pass a SOAPMessageContext in this case, some providers may provide extensions that allow other message types and MessageContext types to be used. A Handler implementation should be ready to accept and ignore message types which it does not understand.

A Handler implementation must use the MessageContext to pass information to other Handler implementations in the same Handler chain and, in the case of the JAX-RPC service endpoint, to the Service Implementation Bean. A container is not required to use the same thread for invoking each Handler or for invoking the Service Implementation Bean.

A Handler may access the `env-entry`s of the component it is associated with by using JNDI to lookup an appropriate subcontext of `java:comp/env`. Access to the `java:comp/env` contexts must be supported from the `init()` and `handle<action>()` methods. Access may not be supported within the `destroy()` method.

A Handler may access the complete SOAP message and can process both SOAP header blocks and body if the handle<action>() method is passed a SOAPMessageContext.

A SOAPMessageContext Handler may add or remove headers from the SOAP message. A SOAPMessageContext Handler may modify the header of a SOAP message if it is not mapped to a parameter or if the modification does not change value type of the parameter if it is mapped to a parameter. A Handler may modify part values of a message if the modification does not change the value type.

A Handler may access transactional resources in a local transaction mode.

Handlers that define application specific headers should declare the header schema in the WSDL document for the component they are associated with, but are not required to do so.

6.2.4 Container Provider Responsibilities

A Handler chain is processed according to the JAX-RPC specification section 12.2.2. The process order defaults to the order the handlers are defined in the deployment descriptor and follow the JAX-RPC specification section 12.1.4 processing order.

A container is required to provide an instance of a `java.util.Map` object in the `HandlerInfo` instance. The `HandlerInfo.getHeaders()` method must return the set of `soap-header`s defined in the deployment descriptor. The Map object must provide access to each of the Handler's `init-param` name/value pairs declared in the deployment descriptor as `java.lang.String` values. The container must provide a unique `HandlerInfo` instance and `Map` config instance for each `Handler` instance. A unique Handler instance must be provider for each Port component declared in the deployment descriptor.

The container must call the `init()` method within the context of a Port component's environment. The container must ensure the Port component's `env-entry`s are setup for the `init` method to access.

The container must provide a `MessageContext` type unique to the request type. For example, the container must provide a `SOAPMessageContext` to the `handle<action>()` methods of a Handler in a handler chain when processing a SOAP request. The `SOAPMessageContext` must contain the complete SOAP message.

The container must share the same `MessageContext` instance across all Handler instances and the target endpoint that are invoked during a single request and response or fault processing on a specific node.

The container must setup the Port component's execution environment before invoking the `handle<action>()` methods of a handler chain. Handlers run under the same execution environment as the Port component's business methods. This is required so that handlers have access to the Port component's `java:comp/env` context.

A container provider is not required to support Handlers for Port component's that run in the EJB container. It is expected this will be required when this specification is included in J2EE 1.4.

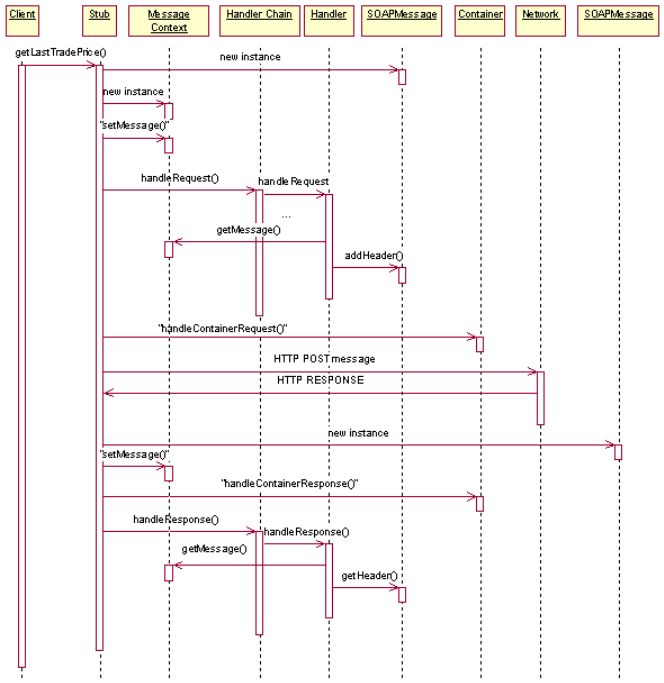
6.3 Packaging

A developer is required to package, either by containment or reference, the Handler class and its dependent classes in the module with the deployment descriptor information that references the Handler classes. A developer is responsible for defining the handler chain information in the deployment descriptor.

6.4 Object Interaction Diagrams

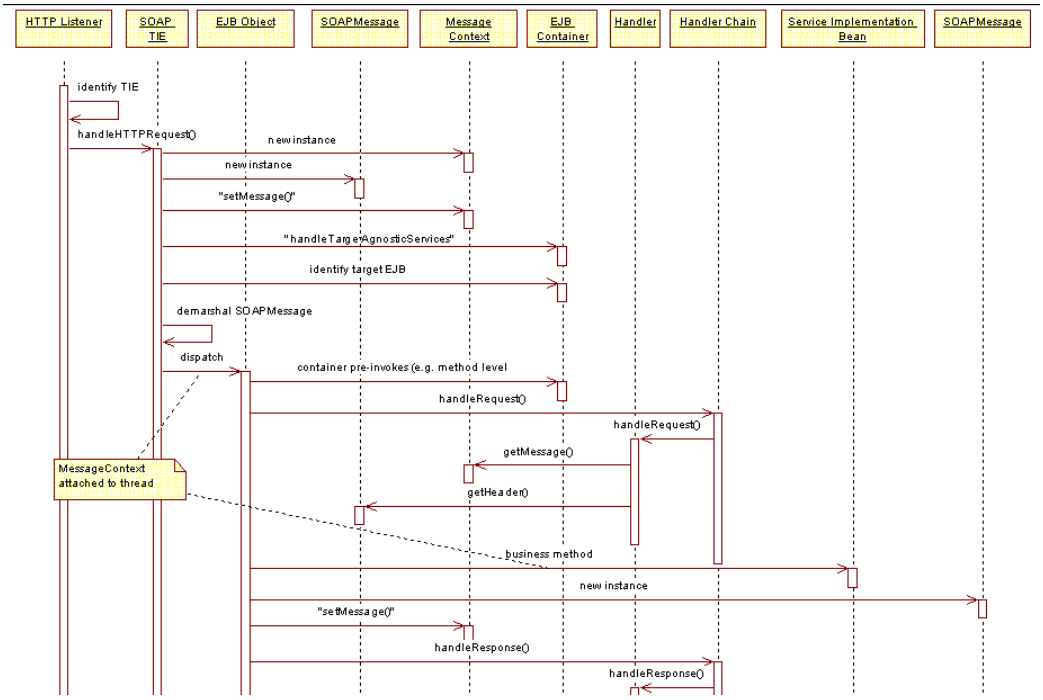
This section contains object interaction diagrams for handler processing. In general, the interaction diagrams are meant to be illustrative.

6.4.1 Client Web service method access

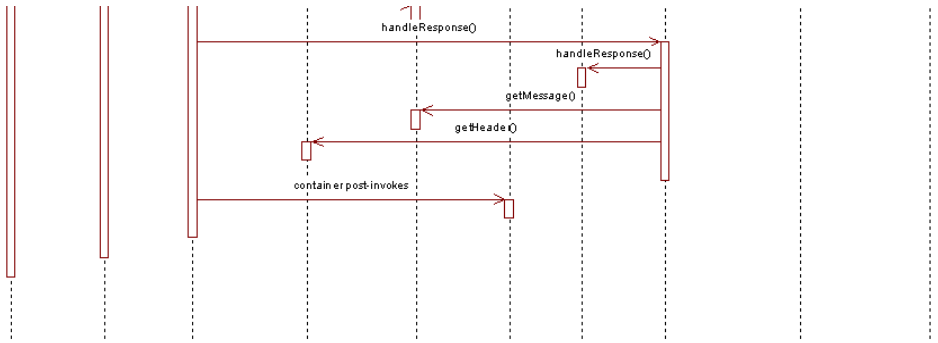


• Figure 8 Client method invoke handler OID

6.4.2 EJB Web service method invocation



• Figure 9 EJB Web service method invocation handler processing part 1



- Figure 10 EJB Web service method invocation handler processing part 2

7 Deployment Descriptors

This chapter describes the various deployment descriptors used for Web Services for J2EE and the roles responsible for defining the information within the deployment descriptors.

7.1 Web Services Deployment Descriptor

This section defines the content of the webservices.xml file, location within modules, roles and responsibilities, and the format.

7.1.1 Overview

The `webservices.xml` deployment descriptor file defines the set of Web services that are to be deployed in a Web Services for J2EE enabled container. The packaging of the `webservices.xml` deployment descriptor file is defined in sections 5.4.1 and 5.4.2. Web services are defined by WSDL documents as described by section 3.2. The deployment descriptor defines the WSDL port to Port component relationship. Port components are defined in Chapter 5.

7.1.2 Developer responsibilities

The developer is responsible not only for the implementation of a Web service, but also for declaring its deployment characteristics. The deployment characteristics are defined in both the module specific deployment descriptor and the webservices.xml deployment descriptor. Service Implementations using a stateless session bean must be defined in the `ejb-jar.xml` deployment descriptor file using the `session` element. Service Implementations using a JAX-RPC Service Endpoint must be defined in the `web.xml` deployment descriptor file using the `servlet-class` element. See the Enterprise JavaBeans and Servlet specifications for additional details on developer requirements for defining deployment descriptors. The developer is also required to provide structural information that defines the Port components within the `webservices.xml` deployment descriptor file. The developer is responsible for providing the set of WSDL documents that describe the Web services to be deployed, the Java classes that represent the Web services, and the mapping that correlates the two.

The developer is responsible for providing the following information in the `webservices.xml` deployment descriptor:

- **Port's name.** A logical name for the port must be specified by the developer using the `port-component-name` element. This name bears no relationship to the WSDL port name. This name must be unique amongst all port component names in a module.
- **Port's bean class.** The developer declares the implementation of the Web service using the `service-impl-bean` element of the deployment descriptor. The bean declared in this element must refer to a class that implements the methods of the Port's Service Endpoint Interface. This element allows a choice of implementations. For a JAX-RPC Service Endpoint, the `servlet-link` element associates the `port-component` with a JAX-RPC Service Endpoint class defined in the `web.xml` by the `servlet-class` element. For a stateless session bean implementation, the `ejb-link` element associates the `port-component` with a `session` element in the `ejb-jar.xml`. The `ejb-link` element may not refer to a `session` element defined in another module. A servlet must only be linked to by a single port-component. A session EJB must only be linked to by a single port-component.

- **Port's Service Endpoint Interface.** The developer must specify the fully qualified class name of the Service Endpoint Interface in the `service-endpoint-interface` element. The Service Endpoint Interface requirements may be found in section 5.3.1.
- **Port's WSDL definition.** The `wsdl-file` element specifies a location of the WSDL description of a set of Web services. The location is relative to the root of the module and must be specified by the developer.
- **Port's QName.** In addition to specifying the WSDL document, the developer must also specify the WSDL port QName in the wsdl-`port` element for each Port defined in the deployment descriptor.
- **JAX-RPC Mapping.** The developer must specify the correlation of the WSDL definition to the interfaces using the `jaxrpc-mapping-file` element. The requirements for specifying information in the `jaxrpc-mapping-file` are covered in section 7.3. The same mapping file must be used for all interfaces associated with a `wsdl-file`.
- **Handlers.** A developer may optionally specify handlers associated with the `port-component` using the `handler` element.
- **Servlet Mapping.** A developer may optionally specify a `servlet-mapping,` in the `web.xml` deployment descriptor, for a JAX-RPC Service Endpoint. No more than one `servlet-mapping` may be specified for a `servlet` that is linked to by a `port-component`. The `url-pattern` of the `servlet-mapping` must be an exact match pattern (i.e. it must not contain an asterisk ("*")).

Note that if the WSDL specifies an address statement within the port, its URI address is ignored. This address is generated and replaced during the deployment process in the deployed WSDL.

See also the developer requirements defined in section 7.2.2.

7.1.3 Assembler responsibilities

The assembler's responsibilities for Web Services for J2EE are an extension of the assembler responsibilities as defined by the Enterprise JavaBeans, Servlet, and J2EE specifications. The assembler creates a deployable artifact by composing multiple modules, resolving cross-module dependencies, and producing an EAR file.

The assembler may modify any of the following information that has been specified by the developer in the `webservices.xml` deployment descriptor file:

- **Description fields.** The assembler may change existing or create new `description` elements.
- **Handlers.** The assembler may change values of existing `param-value` elements, may add new `init-param` elements, may change or add `soap-header` elements, may change or add `soap-role` elements, or may add new `handler` elements.

See also the assembler responsibilities defined in section7.2.3.

7.1.4 Deployer responsibilities

The deployer responsibilities are defined by the J2EE, Enterprise JavaBeans, and Servlet specifications.

In addition, the deployer must resolve the following information:

- where published WSDL definitions are placed. The deployer must publish every `webservice-description` `wsdl-file` with the correct port address attribute value to access the service.
- the value of the port address attribute for deployed services.

7.1.5 Web Services Deployment Descriptor DTD

This is the DTD for the Web service deployment descriptor:

```
<!-- Last updated: 09/21/2002 14:49
Copyright 2002 IBM Corporation
All web service deployment descriptors must use the following
declaration:
<!DOCTYPE webservices PUBLIC "-//IBM Corporation, Inc.//DTD J2EE Web
services 1.0//EN"
"http://www.ibm.com/webservices/dtd/j2ee_web_services_1_0.dtd">
-->

<!--
The webservices element is the root element for the web services
deployment descriptor.  It specifies the set of Web service descriptions
that are to be deployed into the J2EE Application Server and the
dependencies they have on container resources and services.

Used in: webservices.xml
-->
<!ELEMENT webservices (description?, display-name?, small-icon?,
      large-icon?, webservice-description+) >

<!--
The description element is used by the module producer to provide
text describing the parent element.

The description element should include any information that the
module producer wants to provide to the consumer of the module
(i.e. to the Deployer). Typically, the tools used by the module
consumer will display the description when processing the parent
element that contains the description.

Used in: port-component, webservice-description, webservices
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
displayed by tools. The display name need not be unique.

Used in: port-component, webservice-description and webservices

Example:
      <display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>

<!--
The ejb-link element is used in the service-impl-bean element
to specify that a Service Implementation Bean is defined as a Web
Service Endpoint.

The value of the ejb-link element must be the ejb-name of an enterprise
bean in the same ejb-jar file.

Used in: service-impl-bean

Examples:
      <ejb-link>EmployeeRecord</ejb-link>
      <ejb-link>../products/product.jar#ProductEJB</ejb-link>
```

```
-->
<!ELEMENT ejb-link (#PCDATA)>

<!--
Declares the handler for a port-component. Handlers can access the
init-param name/value pairs using the HandlerInfo interface.

Used in: port-component
-->
<!ELEMENT handler (description?, display-name?, small-icon?,
      large-icon?, handler-name, handler-class, init-param*,
      soap-header*, soap-role*)>

<!--
Defines the name of the handler. The name must be unique within the
module.
-->
<!ELEMENT handler-name (#PCDATA)>

<!--
Defines a fully qualified class name for the handler implementation.
-->
<!ELEMENT handler-class (#PCDATA)>

<!--
The init-param element contains a name/value pair as an
initialization param of the servlet
Used in: filter, servlet
-->
<!ELEMENT init-param (param-name, param-value,
      description?)>

<!--
The jaxrpc-mapping-file element contains the name of a file that
describes the JAX-RPC mapping between the Java interaces used by
the application and the WSDL description in the wsdl-file.  The
file name is a relative path within the module.

Used in: webservice-description
-->
<!ELEMENT jaxrpc-mapping-file (#PCDATA)>

<!--
The localpart element indicates the local part of a QNAME.

Used in: soap-header, wsdl-port
-->
<!ELEMENT localpart (#PCDATA)>

<!--
The namespaceURI element indicates a URI.

Used in: soap-header, wsdl-port
-->
<!ELEMENT namespaceURI (#PCDATA)>

<!--
The param-name element contains the name of a parameter. Each
parameter name must be unique in the web application.
Used in: context-param, init-param
-->
<!ELEMENT param-name (#PCDATA)>

<!--
```

```
The param-value element contains the value of a parameter.
Used in: context-param, init-param
-->
<!ELEMENT param-value (#PCDATA)>

<!--
The large-icon element contains the name of a file containing a large
(32 x 32) icon image. The file name is relative path within the
ejb-jar file.

The image must be either in the JPEG or GIF format.
The icon can be used by tools.

Example:
      <large-icon>employee-service-icon32x32.jpg</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The port-component element associates a WSDL port with a Web service
interface and implementation.  It defines the name of
the port as a component, optional description, optional display
name, optional iconic representations, WSDL port QName, Service
Endpoint Interface, Service Implementation Bean.

Used in: webservices
-->
<!ELEMENT port-component (description?, display-name?, small-icon?,
      large-icon?, port-component-name, wsdl-port,
      service-endpoint-interface, service-impl-bean, handler*)>

<!--
The port-component-name element specifies a port component's name.
This name is assigned by the module producer to name the service
implementation bean in the module's deployment descriptor. The name
must be unique among the port component names defined in the same
module.

Used in: port-component

Example:
      <port-component-name>EmployeeService</port-component-name>
-->
<!ELEMENT port-component-name (#PCDATA)>

<!--
The service-endpoint-interface element contains the fully-qualified
name of the port component's Service Endpoint Interface.

Used in: port-component

Example:
      <service-endpoint-
interface>com.wombat.empl.EmployeeService</service-endpoint-interface>
-->
<!ELEMENT service-endpoint-interface (#PCDATA)>

<!--
The service-impl-bean element defines the Web service implementation.
A service implementation can be an EJB bean class or JAX-RPC web
component.  Existing EJB implementations are exposed as a Web service
using an ejb-link.

Used in: port-component
```

```
-->
<!ELEMENT service-impl-bean (ejb-link | servlet-link)>

<!--
The servlet-link element is used in the service-impl-bean element
to specify that a Service Implementation Bean is defined as a
JAX-RPC Service Endpoint.

The value of the servlet-link element must be the servlet-name of
a JAX-RPC Service Endpoint in the same WAR file.

Used in: service-impl-bean

Example:
      <servlet-link>StockQuoteService</servlet-link>
-->
<!ELEMENT servlet-link (#PCDATA)>

<!--
The small-icon element contains the name of a file containing a small
(16 x 16) icon image. The file name is relative path within the
ejb-jar file.

The image must be either in the JPEG or GIF format.
The icon can be used by tools.

Example:
      <small-icon>employee-service-icon16x16.jpg</small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
Defines the QName of a SOAP header that will be processed by the
handler.
-->
<!ELEMENT soap-header (namespaceURI, localpart)>

<!--
The soap-role element contains a SOAP actor definition that the
Handler will play as a role.
-->
<!ELEMENT soap-role (#PCDATA)>

<!--
The webservice-description element defines a WSDL document file
and the set of Port components associated with the WSDL ports
defined in the WSDL document.  There may be multiple
webservice-descriptions defined within a module.

All WSDL file ports must have a corresponding port-component element
defined.

Used in: webservices
-->
<!ELEMENT webservice-description (description?, display-name?,
      small-icon?, large-icon?, webservice-description-name, wsdl-file,
      jaxrpc-mapping-file, port-component+)>

<!--
The webservice-description-name identifies the collection of
port-components associated with a WSDL file and JAX-RPC mapping. The
name must be unique within the deployment descriptor.

Used in: webservice-description
```

```
-->
<!ELEMENT webservice-description-name (#PCDATA)>

<!--
The wsdl-file element contains the name of a WSDL file in the module.
The file name is a relative path within the module.
-->
<!ELEMENT wsdl-file (#PCDATA)>

<!--
Defines the name space and local name part of the WSDL port QName.
-->
<!ELEMENT wsdl-port (namespaceURI, localpart)>

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard EJB deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tools-specific files to the
information in the standard deployment descriptor.
The EJB architecture does not allow the tools to add the non-standard
information into the EJB deployment descriptor.
-->
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST handler id ID #IMPLIED>
<!ATTLIST handler-class id ID #IMPLIED>
<!ATTLIST handler-name id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST jaxrpc-mapping-file id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST localpart id ID #IMPLIED>
<!ATTLIST namespaceURI id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST port-component id ID #IMPLIED>
<!ATTLIST port-component-name id ID #IMPLIED>
<!ATTLIST service-endpoint-interface id ID #IMPLIED>
<!ATTLIST service-impl-bean id ID #IMPLIED>
<!ATTLIST servlet-link id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST soap-header id ID #IMPLIED>
<!ATTLIST soap-role id ID #IMPLIED>
<!ATTLIST webservices id ID #IMPLIED>
<!ATTLIST webservice-description id ID #IMPLIED>
<!ATTLIST webservice-description-name id ID #IMPLIED>
<!ATTLIST wsdl-file id ID #IMPLIED>
<!ATTLIST wsdl-port id ID #IMPLIED>
```

7.2 Web Service Client Deployment Descriptor

This section defines the function of the `webservicesclient.xml` file, location within modules, roles and responsibilities, and the format.

7.2.1 Overview

The `webservicesclient.xml` deployment descriptor contains service reference entries. These entries declare references to Web services used by a J2EE component in the web, EJB, or application client container. If the Web services client is a J2EE component, then it uses a logical name for the Web service

called a service reference to look up the service. Any component that uses a Web service reference must declare a dependency on the Web service reference in a `webservicesclient.xml` deployment descriptor file. The webservicesclient.xml file is packaged in the same directory as the deployment descriptor for the module. For WAR modules, it is packaged in the WEB-INF directory. For EJB JARs and application client modules, it is packaged in the META-INF directory.

7.2.2 Developer responsibilities

The developer is responsible for defining a `service-ref` for each Web service a component within the module wants to reference. This includes the following information:

- **Service Reference Name.** This defines a logical name for the reference that is used in the client source code. It is recommended, but not required that the name begin with `service/`.
- **Service type:** The `service-interface` element defines the fully qualified name of the JAX-RPC Service Interface class returned by the JNDI lookup.
- **Ports.** The developer declares requirements for container managed port resolution using the port-`component-ref` element. The `port-component-ref` elements are resolved to a WSDL port by the container. See Chapter 4 for a discussion of container managed port access.
- **Scope.** If the service reference is being defined within an EJB-JAR, the `service-ref` elements must be defined within a `component-scoped-refs` element so that the `service-ref` can be associated with a particular EJB. The association with the EJB is defined by the `component-name` element.

The developer may specify the following information:

- **WSDL definition.** The `wsdl-file` element specifies a location of the WSDL description of the service. The location is relative to the root of the module. The WSDL description may be a partial WSDL, but must at least include the portType and binding elements. The WSDL description provided by the developer is considered a template that must be preserved by the assembly/deployment process. In other words, the WSDL description contains a declaration of the application's dependency on portTypes, bindings, and QNames. The WSDL document must be fully specified, including the service and port elements, if the application is dependent on port QNames (e.g. uses the `Service.getPort(QName,Class)` method). The developer must specify the `wsdl-file` if any of the Service methods declared in section 4.2.2.4 or 4.2.2.5 are used.
- **Service Port.** If the specified `wsdl-file` has more than one `service` element, the developer must specify the `service-qname`.
- **JAX-RPC Mapping.** The developer specifies the correlation of the WSDL definition to the interfaces using the `jaxrpc-mapping-file` element. The location is relative to the root of the module. The same mapping file must be used for all interfaces associated with a `wsdl-file`. The developer must specify the `jaxrpc-mapping-file` if the `wsdl-file` is specified.
- **Handlers.** A developer may optionally specify handlers associated with the `service-ref` using the `handler` element.

7.2.3 Assembler responsibilities

In addition to the responsibilities defined within the J2EE specification, the assembler may define the following information:

- **Binding of service references**. The assembler may link a Web service reference to a component within the J2EE application unit using the `port-component-link` element. It is the

assembler's responsibility to ensure there are no detailed differences in the SEI and target bindings that would cause stub generation or runtime problems.

The assembler may modify any of the following information that has been specified by the developer in the `webservicesclient.xml` deployment descriptor file:

- **Description fields.** The assembler may change existing or create new `description` elements.
- **Handlers.** The assembler may change values of existing `param-value` elements, may add new `init-param` elements, may change or add `soap-header` elements, may change or add `soap-role` elements, or may add new `handler` elements.
- **WSDL definition.** The assembler may replace the WSDL definition with a new WSDL that resolves missing service and port elements or missing port address attributes. The assembler may update the port address attribute.

7.2.4 Deployer responsibilities

In addition to the normal duties a J2EE deployer platform role has, the deployer must also provide deploy time binding information to resolve the WSDL document to be used for each `service-ref`. If a partial WSDL document was specified and `service` and `port` elements are needed by a vendor to resolve the binding, they may be generated. The deployer is also responsible for providing deploy time binding information to resolve port access declared by the `port-component-ref` element.

7.2.5 Web Services Client Deployment Descriptor DTD

DTD for the webservicesclient.xml Deployment Descriptor:

```
<!-- Last updated: 09/21/2002 14:50
Copyright 2002 IBM Corporation
All Web service client deployment descriptors must use the following
declaration:

<!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.d
td">
-->

<!--
The webservicesclient element is the top level element for service
references.
-->
<!ELEMENT webservicesclient (service-ref+|component-scoped-refs+)>

<!--
The component-name element defines a link to a component name such
as the ejb-name in the module deployment descriptor.  It's value
must exist in the module level deployment descriptor.

Used in: component-scoped-refs
-->
<!ELEMENT component-name (#PCDATA)>

<!--
The component-scoped-refs element defines service references that
are scoped to a particular component of a module.  Not all modules
support component scoping.

Used in: webservicesclient
-->
```

```
<!ELEMENT component-scoped-refs (component-name, service-ref+)>

<!--
The description element gives the deployer a textual description
of the Web service.

Used in: service-ref
-->
<!ELEMENT description (#PCDATA)>

<!--
The display-name element contains a short name that is intended to be
displayed by tools. The display name need not be unique.

Used in: port-component and webservices

Example:
      <display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>

<!--
Declares the handler for a port-component. Handlers can access the
init-param name/value pairs using the HandlerInfo interface. If
port-name is not specified, the handler is assumed to be associated
with all ports of the service.

Used in: service-ref
-->
<!ELEMENT handler (description?, display-name?, small-icon?,
      large-icon?, handler-name, handler-class, init-param*,
      soap-header*, soap-role*, port-name*)>

<!--
Defines a fully qualified class name for the handler implementation.
-->
<!ELEMENT handler-class (#PCDATA)>

<!--
Defines the name of the handler. The name must be unique within the
module.
-->
<!ELEMENT handler-name (#PCDATA)>

<!--
The init-param element contains a name/value pair as an
initialization param of the handler.

Used in: handler
-->
<!ELEMENT init-param (param-name, param-value,
      description?)>

<!--
The jaxrpc-mapping-file element contains the name of a file that
describes the JAX-RPC mapping between the Java interfaces used by
the application and the WSDL description in the wsdl-file.  The
file name is a relative path within the module file.

Used in: webservice-description
-->
<!ELEMENT jaxrpc-mapping-file (#PCDATA)>

<!--
```

```
The large-icon element contains the name of a file containing a large
(32 x 32) icon image. The file name is relative path within the
module file.

The image must be either in the JPEG or GIF format.
The icon can be used by tools.

Example:
      <large-icon>employee-service-icon32x32.jpg</large-icon>
-->
<!ELEMENT large-icon (#PCDATA)>

<!--
The localpart element indicates the local part of a QNAME.

Used in: service-qname, soap-header
-->
<!ELEMENT localpart (#PCDATA)>

<!--
The namespaceURI element indicates a URI.

Used in: service-qname, soap-header
-->
<!ELEMENT namespaceURI (#PCDATA)>

<!--
The param-name element contains the name of a parameter. Each
parameter name must be unique in the web application.

Used in: context-param, init-param
-->
<!ELEMENT param-name (#PCDATA)>

<!--
The param-value element contains the value of a parameter.

Used in: context-param, init-param
-->
<!ELEMENT param-value (#PCDATA)>

<!--
The port-component-link element links a port-component-ref to a
specific port-component required to be made available by a service
reference.

The value of a port-component-link must be the port-component-name
of a port-component in the same module or another module in the same
application unit. The syntax for specification follows the syntax
defined for ejb-link in the EJB 2.0 specification.

Used in: port-component-ref
-->
<!ELEMENT port-component-link (#PCDATA)>

<!--
The port-component-ref element declares a client dependency
on the container for resolving a Service Endpoint Interface
to a WSDL port. It optionally associates the Service Endpoint
Interface with a particular port-component. This is only used
by the container for a Service.getPort(Class) method call.

Used in: service-ref
-->
```

```
<!ELEMENT port-component-ref (service-endpoint-interface,
      port-component-link?)>

<!--
The port-name element defines the WSDL port-name that a handler
should be associated with.
-->
<!ELEMENT port-name (#PCDATA)>

<!--
The service-endpoint-interface element defines a fully qualified
Java class that represents the Service Endpoint Interface of a
WSDL port.

Used in: service-ref
-->
<!ELEMENT service-endpoint-interface (#PCDATA)>

<!--
The service-interface element declares the fully qualified class
name of the JAX-RPC Service interface the client depends on.
In most cases the value will be javax.xml.rpc.Service.  A JAX-RPC
generated Service Interface class may also be specified.

Used in: services-ref
-->
<!ELEMENT service-interface (#PCDATA)>

<!-- The service-qname element declares the specific WSDL service
element that is being referred to.  It is not specified if no
wsdl-file is declared.

Used in service-ref
-->
<!ELEMENT service-qname (namespaceURI, localpart)>

<!-- The service-ref element declares a reference to a Web
service. It contains optional description, display name and
icons, a declaration of the required Service interface,
an optional WSDL document location, an optional set
of JAX-RPC mappings, an optional QName for the service element,
an optional set of Service Endpoint Interfaces to be resolved
by the container to a WSDL port, and an optional set of handlers.

Used in: webservicesclient.xml
-->
<!ELEMENT service-ref (description?, display-name?, small-icon?,
      large-icon?, service-ref-name, service-interface, wsdl-file?,
      jaxrpc-mapping-file?, service-qname?, port-component-ref*,
      handler*)>

<!--
The service-ref-name element declares logical name that the
components in the module use to look up the Web service. It
is recommended that all service reference names start with
"service/".

Used in: services-ref
-->
<!ELEMENT service-ref-name (#PCDATA)>

<!--
The small-icon element contains the name of a file containing a small
(16 x 16) icon image. The file name is relative path within the
```

```
module file.

The image must be either in the JPEG or GIF format.
The icon can be used by tools.

Example:
      <small-icon>employee-service-icon16x16.jpg</small-icon>
-->
<!ELEMENT small-icon (#PCDATA)>

<!--
Defines the QName of a SOAP header that will be processed by the
handler.
-->
<!ELEMENT soap-header (namespaceURI, localpart)>

<!--
The soap-role element contains a SOAP actor definition that the
Handler will play as a role.
-->
<!ELEMENT soap-role (#PCDATA)>

<!--
The wsdl-file element contains the URI location of a WSDL file. The
location is relative to the root of the module.

Used in: service-ref
-->
<!ELEMENT wsdl-file (#PCDATA)>

<!ATTLIST component-name id ID #IMPLIED>
<!ATTLIST component-scoped-refs id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST handler id ID #IMPLIED>
<!ATTLIST handler-class id ID #IMPLIED>
<!ATTLIST handler-name id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST jaxrpc-mapping-file id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST localpart id ID #IMPLIED>
<!ATTLIST namespaceURI id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST port-component-link id ID #IMPLIED>
<!ATTLIST port-component-ref id ID #IMPLIED>
<!ATTLIST port-name id ID #IMPLIED>
<!ATTLIST service-endpoint-interface id ID #IMPLIED>
<!ATTLIST service-interface id ID #IMPLIED>
<!ATTLIST service-qname id ID #IMPLIED>
<!ATTLIST service-ref id ID #IMPLIED>
<!ATTLIST service-ref-name id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST soap-header id ID #IMPLIED>
<!ATTLIST soap-role id ID #IMPLIED>
<!ATTLIST webservicesclient id ID #IMPLIED>
<!ATTLIST wsdl-file id ID #IMPLIED>
```

7.3 JAX-RPC Mapping Deployment Descriptor

This section defines the content of the JAX-RPC mapping file, location within modules, roles and responsibilities, and the format.

7.3.1 Overview

The JAX-RPC mapping deployment descriptor has no standard file name, though it is recommended that the file use a .xml suffix. There is a 1-1 correspondence between WSDL files and mapping files within a module. The JAX-RPC mapping deployment descriptor contains information that correlates the mapping between the Java interfaces and WSDL definition. A deployment tool uses this information along with the WSDL file to generate stubs and TIEs for the deployed services and service-refs.

7.3.2 Developer responsibilities

A developer creates the mapping file at the same time that the WSDL and/or Java interfaces are created. A developer may specify only the package-mapping if the following conditions are met:

- The WSDL file must contain exactly one `service` element.
- The `service` element must define exactly one `port`.
- The set of `service` name, `binding` name, `portType` name, and all root WSDL type (e.g. `complexType`, `simpleType`, etc.) names must be unique.
- The `port`'s `binding` must be a soap 1.1 binding with `style="rpc"`; all its operations must specify `use="encoded", encodingStyle="`*<the SOAP 1.1 encoding>*`"` for their input, output, and fault messages and either omit the parts attribute or include it such that all the parts of both the input and output message are mapped to the soap body. Also, no soap headers or header faults can be specified in the `binding`.
- Each `operation` must:
    - Have a unique name (in the context of the portType it belongs to) that follows the Java conventions for method names.
    - Have exactly one input `message`.
    - Have at most one output `message`.
    - Have zero or more `fault` messages.
    - Either have no parameterOrder attribute or the value of that attribute must be a complete listing of all parts in the input message in the order they appear therein.
- Faults must map to an exception such that it:
    - Directly or indirectly inherits from `java.lang.Exception`, but must not inherit from `RuntimeException` nor `RemoteException`.
    - Has at most a single property called "`message`" of type `java.lang.String` with corresponding single `String` argument constructor.
    - Must be SOAP encoded.
- Each input `message` may have 0 or more `part`s
- Each output `message` must have either 0 or 1 `part`s. If present, the `part` must have a name different from that of any parts in the input message.
- Each `part` must be of this form:

```
<part name="..." type="T"/>
```

- Each type *T* must be one of the following valid types:
    - A simple type defined in table 4-1, section 4.2.1 of the JAX-RPC specification.

- A complex type using either the sequence compositor:

```
<xsd:complexType name="T">
   <xsd:sequence>
          <xsd:element name="..." type="Tprime"/>
   </xsd:sequence>
</xsd:complexType>
```

  Or the all compositor:

```
<xsd:complexType name="T">
   <xsd:all>
          <xsd:element name="..." type="Tprime"/>
   </xsd:all>
</xsd:complexType>
```

  In either case, the element declarations can appear one or more times and each type *Tprime* must be valid. All element names are mapped as JavaBeans properties and element names follow the standard JavaBeans property name convention of lower case for the first character and the complexType name follows the Java standard class name conventions of upper case first letter.

- A SOAP array of the form:

```
<xsd:complexType name="...">
   <xsd:restriction base="soapenc:Array"/>
          <xsd:attribute ref="soapenc:arrayType"
                 wsdl:arrayType="Tprime[]"/>
   </xsd:restriction>
</xsd:complexType>
```

  where Tprime is a valid type and is not a SOAP array type.

If the conditions are not met, a full mapping must be specified. There must be a `java-xml-type-mapping` for every root WSDL type. An `exception-mapping` must be created for each WSDL fault. There must be a `service-interface-mapping` for every `service` element in the WSDL file that has a Generated Service Interface used by the developer. There must be a `service-endpoint-interface-mapping` for every combination of portType and binding in the WSDL file. There must be a `package-mapping` for every namespace defined in the WSDL file.

Web Services for J2EE providers may support partial mapping specifications (e.g. not providing a `method-param-parts-mapping` for every method) regardless of the WSDL content by using standard JAX-RPC WSDL to Java mapping rules to resolve the mappings. If mappings are specified, they take precedence over the mapping rules. Such partial mappings are vendor specific and therefore are non-portable.

For INOUT parameters, only the mapping for the input message is required.

The developer must define the `jaxrpc-mapping-file` element of the webservices.xml or webservicesclient.xml deployment descriptor to be the location of the mapping file.

The developer must package the mapping file in the module with the WSDL file.

7.3.3 Assembler responsibilities

The assembler must not change the JAX-RPC Mapping file.

7.3.4 Deployer responsibilities

The deployer uses deployment tools to deploy the services and service-refs contained inside a module. The deployment tool must use the JAX-RPC mapping file to generate stubs and TIEs for the services and service-refs.

7.3.5 JAX-RPC Mapping DTD

```
<!-- Last updated: 09/21/2002 14:48
Copyright 2002 IBM Corporation.
All Web service mapping deployment descriptors must use the following
declaration:
<!DOCTYPE java-wsdl-mapping PUBLIC "-//IBM Corporation, Inc.//DTD J2EE
JAX-RPC mapping 1.0//EN"
"http://www.ibm.com/webservices/dtd/j2ee_jaxrpc_mapping_1_0.dtd">
-->

<!--
The element describes the Java mapping to a known WSDL document.

It contains the mapping between package names and XML namespaces,
WSDL root types and Java artifacts, and the set of mappings for
services.

-->
<!ELEMENT java-wsdl-mapping (package-mapping+,
      java-xml-type-mapping*, exception-mapping*,
      (service-interface-mapping?,
      service-endpoint-interface-mapping+)*)>

<!--
The class-type element is the fully qualified class name of
a Java class.

Used in: java-xml-type-mapping
-->
<!ELEMENT class-type (#PCDATA)>

<!--
The constructor-parameter-order element defines the order
that complexType element values are applied to a Java
exception constructor. Element names are specified for each
parameter of the constructor, including element names of
inherited types if necessary.

Used in: exception-mapping
-->
<!ELEMENT constructor-parameter-order (element-name+)>

<!--
The data-member element is a boolean indicator that a Java
variable is a public data member and not a JavaBeans property.

Used in: variable-mapping
-->
<!ELEMENT data-member EMPTY>

<!--
The element-name element defines the name of a complexType
element name attribute value.

Used in: constructor-parameter-order
```

```
-->
<!ELEMENT element-name (#PCDATA)>

<!--
The exception-mapping element defines the mapping between the
service specific exception types and the wsdl faults.

This element should be interpreted with respect to the
mapping between a method and an operation which provides the
mapping context.

Used in: service-endpoint-method-mapping
-->
<!ELEMENT exception-mapping (exception-type, wsdl-message,
      constructor-parameter-order?)>

<!--
The exception-type element defines Java type of the exception.
It may be a service specific exception.

It must be a fully qualified class name.

Used in: exception-mapping
-->
<!ELEMENT exception-type (#PCDATA)>

<!--
The java-method-name element defines the name of a Java method within
an interface.

Used in: service-endpoint-method-mapping
-->
<!ELEMENT java-method-name (#PCDATA)>

<!--
The java-port-name element is the string to use as the port name in
Java.  It is used in generating the Generated Service Interface method
get<java-port-name>.

Used in: port-mapping
-->
<!ELEMENT java-port-name (#PCDATA)>

<!--
The java-variable-name defines the name of a public data member or
JavaBeans property within a Java class.

Used in: variable-mapping
-->
<!ELEMENT java-variable-name (#PCDATA)>

<!--
The java-xml-type-mapping element contains a class-type that is the
fully qualified name of the Java class, QName of the XML root type,
the WSDL type scope the QName applies to and the set of variable
mappings for each public variable within the Java class.

Used in: java-wsdl-mapping
-->
<!ELEMENT java-xml-type-mapping (class-type, root-type-qname,
      qname-scope, variable-mapping*)>

<!--
The localpart element indicates the local part of a QNAME.
```

```
Used in: wsdl-binding, wsdl-message, wsdl-port-type,
            wsdl-service-name
-->
<!ELEMENT localpart (#PCDATA)>

<!--
The method-param-parts-mapping element defines the mapping between a
Java method parameters and a wsdl-message.

Used in: service-endpoint-method-mapping
-->
<!ELEMENT method-param-parts-mapping (param-position, param-type,
      wsdl-message-mapping)>

<!--
The method-return-value  element defines a fully qualified class name
or void type for the method's return value type.

Used in: wsdl-return-value-mapping
-->
<!ELEMENT method-return-value (#PCDATA)>

<!--
The namespaceURI element indicates a URI.

Used in: package-mapping, wsdl-binding, wsdl-message, wsdl-port-type,
            wsdl-service-name
-->
<!ELEMENT namespaceURI (#PCDATA)>

<!--
The package-mapping indicates the mapping between java-package-name
and XML namespace in the WSDL document.

Used in: java-wsdl-mapping
-->
<!ELEMENT package-mapping (package-type, namespaceURI)>

<!--
The package-type indicates the Java package name. It must be a fully
qualified name.

Used in: package-mapping
-->
<!ELEMENT package-type (#PCDATA)>

<!--
The parameter-mode  element defines the mode of the parameter.
It can have only three values, IN, OUT, INOUT.

Used in: wsdl-message-mapping, wsdl-return-value-mapping
-->
<!ELEMENT parameter-mode (#PCDATA)>

<!--
The param-position element defines the position of a parameter within
a Java method.  It must be an integer starting from 0.

Used in: method-param-parts-mapping
-->
<!ELEMENT param-position (#PCDATA)>

<!--
```

```
The param-type element defines the Java type of a parameter within a
Java method. It must be defined by a fully qualified name of a class.

Used in: method-param-parts-mapping
-->
<!ELEMENT param-type (#PCDATA)>

<!--
The port-mapping defines the mapping of the WSDL port name attribute
to the Java name used to generate the Generated Service Interface
method get<java-name>.

Used in: service-interface-mapping
-->
<!ELEMENT port-mapping (port-name, java-port-name)>

<!--
The port-name is the attribute value of a name attribute of a WSDL
port element.

Used in: port-mapping
-->
<!ELEMENT port-name (#PCDATA)>

<!--
The qname-scope elements scopes the reference of a QName to the WSDL
element type it applies to. The value of qname-scope may be
simpleType, complexType, or element.

Used in: java-xml-type-mapping
-->
<!ELEMENT qname-scope (#PCDATA)>

<!--
The root-type-qname identifies the WSDL QName of an XML type.

Used in: java-xml-type-mapping
-->
<!ELEMENT root-type-qname (namespaceURI, localpart)>

<!--
The service-endpoint-interface element defines the Java type for the
endpoint interface. The name must be a fully qualified class name.

Used in: service-endpoint-interface-mapping
-->
<!ELEMENT service-endpoint-interface (#PCDATA)>

<!--
The service-endpoint-interface-mapping defines a tuple
to specify Service Endpoint Interfaces to
WSDL port types and WSDL bindings.


An interface may be mapped to a port-type and binding multiple
times. This happens rarely.

Used in: java-wsdl-mapping
-->
<!ELEMENT service-endpoint-interface-mapping (
      service-endpoint-interface, wsdl-port-type, wsdl-binding,
      service-endpoint-method-mapping*)>

<!--
```

```
The service-endpoint-method-mapping element defines the mapping of
Java methods to operations (which are not uniquely qualified by
qnames).

The wsdl-operation should be interpreted with respect to the portType
and binding in which this definition is embedded within. See the
definitions for service-endpoint-interface-mapping and
service-interface-mapping to acquire the proper context. The
wrapped-element indicator should only be specified when a WSDL
message wraps an element type.  The wsdl-return-value-mapping is
not specified for one-way operations.

Used in: service-endpoint-interface-mapping
-->
<!ELEMENT service-endpoint-method-mapping (java-method-name,
          wsdl-operation, wrapped-element?, method-param-parts-mapping*,
          wsdl-return-value-mapping?)>

<!--
The service-interface element defines the Java type for the service.
For static services, it is javax.xml.rpc.Service interface. For
generated service, it would be the generated interface name.

The name must be a fully qualified class name.

Used in: service-interface-mapping
-->
<!ELEMENT service-interface (#PCDATA)>

<!--
The service-interface-mapping element defines how a Java type for
the service interface maps to a WSDL service.

Used in: java-wsdl-mapping
-->
<!ELEMENT service-interface-mapping  (service-interface,
      wsdl-service-name, port-mapping*)>

<!--
The soap-header element is a boolean element indicating that
a parameter is mapped to a SOAP header.

Used in: wsdl-message-mapping
-->
<!ELEMENT soap-header EMPTY>

<!--
The variable-mapping element defines the correlation between a
Java class data member or JavaBeans property to an XML element
name of an XML root type. If the data-member element is present,
the Java variable name is a public data member.  If data-member
is not present, the Java variable name is a JavaBeans property.

Used in: java-xml-type-mapping
-->
<!ELEMENT variable-mapping (java-variable-name, data-member?,
      xml-element-name)>

<!--
The wrapped-element element is defined when a WSDL message with
a single part is used to wrap an element type and the element's
name matches the operation name.

Used in: service-endpoint-method-mapping
```

```
-->
<!ELEMENT wrapped-element EMPTY>

<!--
The wsdl-binding element defines the wsdl binding
by a QNAME which uniquely identifies  the binding.

Used in: service-endpoint-interface-mapping
-->
<!ELEMENT wsdl-binding (namespaceURI, localpart)>

<!--
The wsdl-message  element defines a WSDL message by a QNAME.

Used in: wsdl-message-mapping, wsdl-return-value-mapping
-->
<!ELEMENT wsdl-message (namespaceURI, localpart)>

<!--
The wsdl-message-mapping element defines the mapping to a specific
message and its part. Together they define uniquely the mapping for
a specific parameter. Parts within a message context are uniquely
identified with their names.

The parameter-mode is defined by the mapping to indicate whether
the mapping will be IN, OUT, or INOUT..  The presence of the soap-header
element indicates that the parameter is mapped to a soap header only.
When absent, it means that the wsdl-message is mapped to a Java
parameter. The soap headers are interpreted in the order they are
provided in the mapping.

Used in: method-param-parts-mapping
-->
<!ELEMENT wsdl-message-mapping (wsdl-message, wsdl-message-part-name,
      parameter-mode, soap-header?)>

<!--
Interpretation of the wsdl-message-part-name element depends on whether
or not wrapped-element has been defined in the
service-endpoint-method-mapping.  If wrapped-element is not specified,
wsdl-message-part-name defines a WSDL message part. It
should always be interpreter with respect to a wsdl-message element. If
wrapped-element is specified, wsdl-message-part-name refers to an
element
name of the element type.

Used in: wsdl-message-mapping, wsdl-return-value-mapping
-->
<!ELEMENT wsdl-message-part-name (#PCDATA)>

<!--
The wsdl-operation element defines an operation within a WSDL document.
It must be interpreted with respect to a port type.

Used in: service-endpoint-method-mapping
-->
<!ELEMENT wsdl-operation (#PCDATA)>

<!--
The wsdl-port-type element defines the wsdl port type
by a QNAME which uniquely identifies  the port type.

Used in: service-endpoint-interface-mapping
-->
```

```
<!ELEMENT wsdl-port-type (namespaceURI, localpart)>

<!--
The wsdl-return-value-mapping  element defines the mapping for the
method's return value. It defines the mapping to a specific message
and its part.  Together they define uniquely the mapping for a
specific parameter. Parts within a message context are uniquely
identified with their names. The wsdl-message-part-name is not
specified if there is no return value or OUT parameters.

Used in: service-endpoint-method-mapping
-->
<!ELEMENT wsdl-return-value-mapping (method-return-value, wsdl-message,
      wsdl-message-part-name?)>

<!--
The wsdl-service-name element defines the wsdl service name
by a QNAME which uniquely identifies  the service.

Used in: service-interface-mapping
-->
<!ELEMENT wsdl-service-name (namespaceURI, localpart)>

<!--
The xml-element-name element defines name attribute value of a WSDL
element within a root type.

Used in: variable-mapping
-->
<!ELEMENT xml-element-name (#PCDATA)>

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard EJB deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tools-specific files to the
information in the standard deployment descriptor.
The EJB architecture does not allow the tools to add the non-standard
information into the EJB deployment descriptor.
-->
<!ATTLIST class-type id ID #IMPLIED>
<!ATTLIST constructor-parameter-order id ID #IMPLIED>
<!ATTLIST data-member id ID #IMPLIED>
<!ATTLIST element-name id ID #IMPLIED>
<!ATTLIST exception-mapping id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST java-method-name id ID #IMPLIED>
<!ATTLIST java-port-name id ID #IMPLIED>
<!ATTLIST java-variable-name id ID #IMPLIED>
<!ATTLIST java-wsdl-mapping id ID #IMPLIED>
<!ATTLIST java-xml-type-mapping id ID #IMPLIED>
<!ATTLIST localpart id ID #IMPLIED>
<!ATTLIST method-param-parts-mapping id ID #IMPLIED>
<!ATTLIST method-return-value id ID #IMPLIED>
<!ATTLIST namespaceURI id ID #IMPLIED>
<!ATTLIST package-mapping id ID #IMPLIED>
<!ATTLIST package-type id ID #IMPLIED>
<!ATTLIST parameter-mode id ID #IMPLIED>
<!ATTLIST param-position id ID #IMPLIED>
<!ATTLIST param-type id ID #IMPLIED>
<!ATTLIST port-mapping id ID #IMPLIED>
<!ATTLIST port-name id ID #IMPLIED>
<!ATTLIST qname-scope id ID #IMPLIED>
<!ATTLIST root-type-qname id ID #IMPLIED>
```

```
<!ATTLIST service-endpoint-interface id ID #IMPLIED>
<!ATTLIST service-endpoint-interface-mapping id ID #IMPLIED>
<!ATTLIST service-endpoint-method-mapping id ID #IMPLIED>
<!ATTLIST service-interface id ID #IMPLIED>
<!ATTLIST service-interface-mapping id ID #IMPLIED>
<!ATTLIST soap-header id ID #IMPLIED>
<!ATTLIST variable-mapping id ID #IMPLIED>
<!ATTLIST wrapped-element id ID #IMPLIED>
<!ATTLIST wsdl-binding id ID #IMPLIED>
<!ATTLIST wsdl-message id ID #IMPLIED>
<!ATTLIST wsdl-message-mapping id ID #IMPLIED>
<!ATTLIST wsdl-message-part-name id ID #IMPLIED>
<!ATTLIST wsdl-operation id ID #IMPLIED>
<!ATTLIST wsdl-port-type id ID #IMPLIED>
<!ATTLIST wsdl-return-value-mapping id ID #IMPLIED>
<!ATTLIST wsdl-service-name id ID #IMPLIED>
<!ATTLIST xml-element-name id ID #IMPLIED>
```

8 Deployment

This chapter defines the deployment process requirements and responsibilities. Deployment tasks are handled by the J2EE deployer platform role using tools typically provided by the Web Services for J2EE product provider. This includes the generation of container specific classes for the Web services and Web service references, configuration of the server's SOAP request listeners for each port, publication and location of Web services, as well as the normal responsibilities defined by the J2EE specification.

8.1 Overview

This section describes an illustrative process of deployment for Web Services for J2EE. The process itself is not required, but there are certain requirements that deployment must meet which are detailed in later sections of this chapter. This process assumes that there are two general phases for deployment. The first phase maps Web services into standard J2EE artifacts and the second phase is standard J2EE deployment.

Deployment starts with a service enabled application or module. The deployer uses a deployment tool to start the deployment process. In general, the deployment tool validates the content as a correctly assembled deployment artifact, collects binding information from the deployer, deploys the components and Web services defined within the modules, publishes the WSDL documents representing the deployed Web services, deploys any clients using Web services, configures the server and starts the application.

The deployment tool starts the deployment process by examining the deployable artifact and determining which modules are Web service enabled by looking for a webservices.xml deployment descriptor file contained within the module. Deployment of services occurs before resolution of service references. This is done to allow deployment to update the WSDL port addresses before the service references to them are processed.

Validation of the artifact packaging is performed to ensure that:

- Every port in every WSDL defined in the Web services deployment descriptor has a corresponding `port-component` element.
- If the Service Implementation Bean is an EJB, the transaction attributes for the methods defined by the SEI do not include Mandatory.
- If the Service Implementation Bean is an EJB, the SEI methods are fully contained by the remote interface.
- JAX-RPC service components are only packaged within a WAR file.
- Stateless session bean Web services are only packaged within an EJB-JAR file.
- The WSDL bindings used by the WSDL ports are supported by the Web Services for J2EE runtime. Bindings that are not supported may be declared within the WSDL if no port uses them.

Deployment of each `port-component` is dependent upon the service implementation and container used. Deployment of a JAX-RPC Service Endpoint requires different handling than deployment of a session bean service.

If the implementation is a JAX-RPC Service Endpoint, a servlet is generated to handle parsing the incoming SOAP request and dispatch it to an instance of the JAX-RPC service component. The generated servlet class is dependent on threading model of the JAX-RPC Service Endpoint. The `web.xml` deployment descriptor is updated to replace the JAX-RPC Service Endpoint class with the generated servlet class. If the JAX-RPC Service Endpoint was specified without a corresponding `servlet-mapping`, the deployment tool generates one. The WSDL port address for the Port component is the combination of the web app `context-root` and servlet `url-pattern`. If the implementation is a stateless session bean, the

deployment tool has a variety of options available to it. In general, the deployment tool generates a servlet to handle parsing the incoming SOAP request, create a remote stateless session EJBObject and dispatch the request to the stateless session EJB. The methods of the SEI are described by the remote interface, the deployment tool can generate a servlet that dispatches to the remote interface of the EJB. How the request is dispatched to the Service Implementation Bean is dependent on the deployment tool and deploy time binding information supplied by the deployer.

The deployment tool must deploy and publish all the ports of all WSDL documents described in the Web services deployment descriptor. The deployment tool updates or generates the WSDL port address for each deployed `port-component`. The updated WSDL documents are then published to a location determined by the deployer. It could be as simple as publishing to a file in the modules containing the deployed services, a URL location representing the deployed services of the server, a UDDI or ebXML registry, or a combination of these. This is required for the next step, which is resolving references to Web services.

For each service reference described in the Web services client deployment descriptors, the deployment tool ensures that the client code can access the Web service. The deployment tool examines the information provided in the client deployment descriptor (the Service interface class, the Service Endpoint Interface class, and WSDL ports the client wants to access) as well as the JAX-RPC mapping information. In general the procedure includes providing an implementation of the JAX-RPC Service interface class declared in the deployment descriptor service reference, generating stubs for all the `service-endpoint-interface` declarations (if generated Stubs are supported and the deployer decides to use them), and binding the Service class implementation into a JNDI namespace. The specifics depend on whether or not the service is declared as a client managed or container managed access.

When client managed port access is used, the deployment tool must provide generated stubs or dynamic proxy access to every port declared within the Web services client deployment descriptor. The choice of generated stub or dynamic proxy is deploy time binding information. The container must provide an implementation for a Generated Service Interface if declared within the deployment descriptor.

When container managed port access to a service is used, the container must provide generated stubs or dynamic proxy access to every port declared within the deployment descriptor. The choice of generated stub or dynamic proxy is deploy time binding information. The deployment descriptor may contain a `port-component-link` to associate the reference not only with the Service Endpoint Implementation, but with the WSDL that defines it.

Once the Web services enabled deployable artifact has been converted into a J2EE deployable artifact, the deployment process continues using normal deployment processes.

8.2 Container Provider requirements

This section details the requirements of the container provider. This includes both the container runtime and the deployment tooling.

8.2.1 Deployment artifacts

A deployment tool must be capable of deploying an EAR file (containing WARs and/or EJB-JARs), WAR file, or EJB-JAR containing Web services and/or Web services references.

8.2.2 Generate Web Service Implementation classes

Generation of any run-time classes the container requires to support a JAX-RPC Service Endpoint or Stateless Session Bean Service Implementation is provider specific. The behavior of the run-time classes must match the deployment descriptor settings of the component. A JAX-RPC Service Endpoint must match the behavior defined by the `<servlet>` element in the `web.xml` deployment descriptor. A Stateless

Session Bean Service Implementation must match the behavior defined by the `<session>` element and the `<assembly-descriptor>` in the `ejb-jar.xml` deployment descriptor.

8.2.3 Generate deployed WSDL

The container must update and/or generate a deployed WSDL document for each declared `wsdl-file` element in the Web services deployment descriptor (webservices.xml). If multiple `wsdl-file` elements refer to the same location, a separate WSDL document must be generated for each.

The WSDL document described by the `wsdl-file` element must contain service and port elements and every `port-component` in the deployment descriptor must have a corresponding WSDL port and vice versa. The deployment tool must update the WSDL port address element to produce a deployed WSDL document. The generated port address information is deployment time binding information. In the case of a `port-component` within a web module, the address is partially constrained by the `context-root` of the web application and partially constructed from the `servlet-mapping` (if specified).

8.2.4 Publishing the deployed WSDL

The deployment tool must publish every deployed WSDL document. The deployed WSDL document may be published to a file, URL, or registry. File and URL publication must be supported by the provider. File publication includes within the generated artifacts of the application. Publication to a registry, such as UDDI or ebXML, is encouraged but is not required.

If publication to a location other than file or URL is supported, then location of a WSDL document containing a service from that location must also be supported. As an example, a Web services deployment descriptor declares a `wsdl-file StockQuoteDescription.xml` and a `port-component` which declares a `port` QName within the WSDL document. When deployed, the port address in `StockQuoteDescription.xml` is updated to the deployed location. This is published to a UDDI registry location. In the same application, a `service-ref` uses a `port-component-link` to refer to the deployed `port-component`. The provider must support locating the deployed WSDL for that port component from the registry it was published to. This support must be available to a deployed client that is not bundled with the application containing the service.

Publishing to at least one location is required. Publishing to multiple locations is allowed, but not required. The choice of where (both location and how many places) to publish is deployment time binding information.

8.2.5 Service and Generated Service Interface implementation

The container must provide an implementation of the JAX-RPC Service Interface. There is no requirement for a Service Implementation to be created during deployment. The container may substitute a Generated Service Interface Implementation for a generic Service Interface Implementation.

The container must provide an implementation of the JAX-RPC Generated Service Interface if the Web services client deployment descriptor defines one. A Generated Service Interface Implementation will typically be provided during deployment.

The Service Interface Implementation must provide a static stub and/or dynamic proxy for all ports declared by the service element in the WSDL description. A container provider must support at least one of static stubs or dynamic proxies, but may provide support for both.

The container must make the required Service Interface Implementation available at the JNDI namespace location `java:comp/env/`*service-ref-name* where *service-ref-name* is the name declared within the Web services client deployment descriptor using the `service-ref-name` element.

8.2.6 Static stub generation

A deployment tool may support generation of static stubs. A container provider must support static stub generation if dynamic proxies are not supported. Static stubs are provider specific and, in general, a developer should avoid packaging them with the application.

Static stubs (and dynamic proxies) must conform to the JAX-RPC specification sections 8.2.1 and 8.2.2.

The container is required to support credential propagation as defined in section 4.2.4 without client code intervention. Whether or not the stub/proxy directly supports this or another part of the container does is out of the scope of this specification.

8.2.7 Type mappings

Support for type mappings is provider specific. There is no means for creating portable type mappings and therefore no means for declaring them or deploying them required by this specification.

8.2.8 Mapping requirements

The deployment tool must use the mapping meta-data requirements defined by the `jaxrpc-mapping-file`. All mappings must be applied before default rules are applied.

8.2.9 Deployment failure conditions

Deployment may fail if:

- The webservices.xml deployment descriptor is invalid
- The WSDL file, JAX-RPC mapping file and deployment descriptor conflict
- The implementation methods and operations conflict
- Any Port component cannot be deployed
- Every port in every WSDL defined in the Web services deployment descriptor doesn't have a corresponding `port-component` element.
- If the Service Implementation Bean is an EJB, the transaction attributes for the methods defined by the SEI include Mandatory.
- If the Service Implementation Bean is an EJB, the SEI methods are not fully contained by the remote interface.
- JAX-RPC service components are not packaged within a WAR file.
- Stateless session bean Web services are not packaged within an EJB-JAR file.
- The WSDL bindings used by the WSDL ports are not supported by the Web Services for J2EE runtime. However, bindings that are not supported may be declared within the WSDL if no port uses them.
- The header QNames returned by a Handler.getHeaders() method are not defined in the WSDL for the port-component the Handler is executing on behalf of.

8.3 Deployer responsibilities

The deployer role is responsible for specifying the deployment time binding information. This may include deployed WSDL port addresses and credential information for requests that do not use a CallbackHandler.

If a `service-ref` contains a `port-component-ref` that contains a `port-component-link`, the deployer should bind the container managed Port for the SEI to the deployed port address of the `port-component` referred to by the `port-component-link`. For example, given a `webservices.xml` file containing:

```
<webservices>
  <webservice-description>
    <webservice-description-name>JoesServices</webservice-description-name>
    <wsdl-file>META-INF/joe.wsdl</wsdl-file>
    <jaxrpc-mapping-file>META-INF/joes_mappings.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>JoePort</port-component-name>
      …
      <service-impl-bean>
        <ejb-link>JoeEJB</ejb-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
</webservices>
```

and a webservicesclient.xml containing:

```
<webservicesclient>
  <service-ref>
    <service-ref-name>service/Joe</service-ref-name>
    <service-interface>javax.xml.rpc.Service</service-interface>
    <wsdl-file>WEB-INF/joe.wsdl</wsdl-file>
    …
    <port-component-ref>
      <service-endpoint-interface>sample.Joe</service-endpoint-interface>
      <port-component-link>JoePort</port-component-link>
    </port-component-ref>
  </service-ref>
</webservicesclient>
```

During deployment, the deployer must provide a binding for the port address of the JoePort port-component. This port address must be defined in the published WSDL for JoesServices. The deployer must also provide a binding for container managed port access to the sample.Joe Service Endpoint Interface. This should be the same binding used for the port address of the JoePort port-component.

When providing a binding for a `port-component-ref`, the deployer must ensure that the `port-component-ref` is compatible with the Port being bound to.

9 Security

This section defines the security requirements for Web Services for J2EE. A conceptual overview of security and how it applies to Web services is covered in the Concepts section. The Goals section defines what this specification attempts to address and the Specification section covers the requirements.

9.1 Concepts

The Web services security challenge is to understand and assess the risk involved in securing a web based service today and at the same time to track emerging standards and understand how they will be deployed to offset the risk in the future. Any security model must illustrate how data can flow through an application and network topology to meet the requirements defined by the business without exposing the data to undue risk. A Web services security model should support protocol independent declarative security policies that Web Service for J2EE providers can enforce, and descriptive security policies attached to the service definitions that clients can use in order to securely access the service.

The five security requirements that need to be addressed to assure the safety of information exchange are:

- **Authentication** – the verification of the claimant's entitlements to use the claimed identity and/or privilege set.
- **Authorization** – the granting of authority to an identity to perform certain actions on resources
- **Integrity** – the assurance that the message was not modified accidentally or deliberately in transit.
- **Confidentiality** – the guarantee that the contents of the message are not disclosed to unauthorized individuals.
- **Non-repudiation** – the guarantee that the sender of the message cannot deny that the sender has sent it. This request also implies message origin authentication.

The risks associated with these requirements can be avoided with a combination of various existing and emerging technologies and standards in J2EE environments. There are fundamental business reasons underlying the existence of various security mechanisms to mitigate the various security risks outlined above. The authentication of the entity is necessary. This helps provide access based on the identity of the caller of the Web service. The business reason for data integrity is so that each party in a transaction can have confidence in the business transaction. It's also a business-legal issue to have an audit trail and some evidence of non-repudiation to address liability issues. And more and more businesses are becoming aware of the internal threats to their applications by employees or others *inside* the firewall. Some business transactions require that confidentiality be provided on a service invocation or its data (like credit card numbers). There is also the need for businesses on the Internet to protect themselves from denial of service attacks being mounted. This is the environment in which we need to assert a security service model.

9.1.1 Authentication

Since the Web services architecture builds on existing component technologies, intra-enterprise authentication is no different than today's approaches. In order for two or more parties to communicate securely they may need to exchange security credentials. Web service's security is used to exchange many different types of credentials. A credential represents the authenticity of the identity it is associated with e.g., Kerberos ticket. A credential can be validated to verify the authenticity and the identity can then be inferred from the credential.

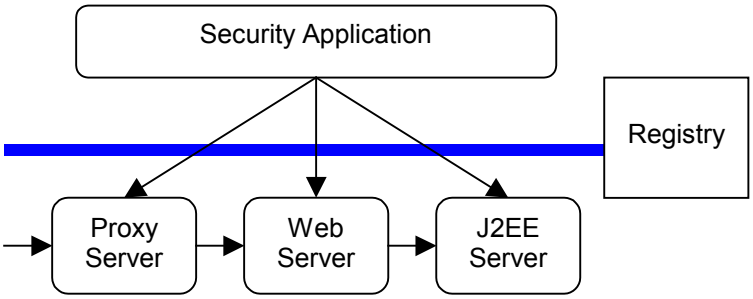
When two parties communicate, it is also important for the sender to understand the security requirements of the target service. This helps the sender to provide necessary credentials along with the request. Alternatively,

the target may challenge the sender for necessary credential (similar to how HTTP servers challenge the HTTP clients).

In the future, it is expected that message level security mechanisms will be supported. Using that approach, credentials can be propagated along with a message and independent of the underlying transport protocols. Similarly, confidentiality and integrity of a message can be ensured using message level protection. Message level security support would help address end-to-end security requirements so that requests can traverse through multiple network layers, topologies and intermediaries in a secure fashion independent of the underlying protocol.

In the future, it is also anticipated that in order for client applications to determine the level of security expected by a Web service and the expected type of credential, the information about the authentication policy will be included in or available through the service definition (WSDL). Based on that service definition, a client provides appropriate credentials. If the container has policies for the service, then they must be referenced and used.

- Figure 11 security flow overview



Consider a scenario where incoming SOAP/WSDL messages flow over HTTP(S). The figure above provides a simple overview of the security flow. Enterprise Web sites rely on the J2EE Server support for the authentication models. The site also relies on a Proxy Server's support for security. In these scenarios, authentication occurs before the J2EE Server receives the request. In these cases, a Proxy Server or Web Server forwards authentication credentials into the J2EE Application Server. The J2EE application server handles the request similar to how it handles other HTTP requests. J2EE application server can handle the request similar to how it handles other HTTP requests.

Two forms of authentication are available for use within Web Services for J2EE based on existing J2EE functionality. These are HTTP BASIC-AUTH and Symmetric HTTP, which are defined by the Servlet specification.

Using the authentication models above, the container can also perform a credential mapping of incoming credentials at any point along the execution path. The mapping converts the external user credentials into a credential used within a specific security domain, for example by using Kerberos or other imbedded third party model.

In addition to J2EE security model for credential propagation, it may be beneficial to carry identity information within SOAP message itself (e.g., as a SOAP header). This can help address situations where Web services need to be supported where inherent security support of underlying transport and protocols may not be sufficient (e.g., JMS). JSR109 does not require any support for credential propagation within SOAP messages and considers this functionality as future work.

9.1.2 Authorization

In an enterprise security model, each application server and middleware element performs authorization for its resources (EJBs, Servlets, Queues, Tables, etc.). The J2EE authentication/delegation model ensures that the user identity is available when requests are processed .

On successful authentication, identity of the authenticated user is associated with the request. Based on the identity of the user, authorization decisions are made. This is performed by the J2EE Servers based on the J2EE security model to only allow authorized access to the methods of EJBs and Servlets/JSPs. Authorization to Web services implemented as JAX-RPC Service Endpoints will be based on the servlet/JSP security model.

9.1.3 Integrity and Confidentiality

In general, integrity and confidentiality are based on existing J2EE support such as HTTPS.

Message senders may also want to ensure that a message or parts of a message remain confidential and that it is not modified during transit. When a message requires confidentiality, the sender of the message may encrypt those portions of the message that are to be kept private using XML Encryption. When the integrity of a message is required to be guaranteed, the sender of the message may use XML Digital Signature to ensure that the message is not modified during transit. This specification recommends that J2EE servers use XML Encryption for confidentiality, and XML Digital Signature for integrity but defers to future work to standardize the format and APIs.

9.1.4 Audit

J2EE Servers can optionally write implicit and explicit audit records when processing requests. The middleware flows the user credentials and a correlation ID in an implicit context on all operations. Management tools can gather the multiple logs, merge them and use the correlation information to see all records emitted processing an incoming Web service request. It is recommended that J2EE servers implement support for audit records, but defers to the J2EE to standardize the record formats and APIs to support audit logs.

9.1.5 Non-Repudiation

The combination of Basic Authentication over HTTP/S is widely used in the industry today to ensure confidentiality, authentication and integrity. However, it fails to assure non-repudiation.

It is recommended that J2EE servers implement support for non-repudiation logging, but does not define a standard mechanism to define and support it.

9.2 Goals

The security model for Web services in J2EE application servers should be simple to design and use, ubiquitous, cost effective, based on open standards, extensible, and flexible. The base functionality needs to be able to be used for the construction of a wide variety of security models, security authentication credentials, multiple trust domains and multiple encryption technologies. Therefore, the goals for security include the following:

- Should support protecting Web services using J2EE authorization model.
- Should support propagating authentication information over the protocol binding through which a Web service request is submitted.

- Should support transport level security to ensure confidentiality and integrity of a message request.
- Should be firewall friendly; be able to traverse firewalls without requiring the invention of special protocols.

9.2.1 Assumptions

The following assumptions apply to this chapter:

The server relies on the security infrastructure of the J2EE Application Server.

The Quality of Service (QoS) of a secure Web service container is based on the QoS requirements and functionality of the underlying J2EE application server itself (e.g., integrity).

The server relies on HTTPS and RMI-IIOP over SSL for hop-by-hop confidentiality and integrity .

9.3 Specification

The following sections define the requirements for implementing security for Web Services for J2EE.

9.3.1 Authentication

There are few authentication models to authenticate message senders that are adopted or proposed as standards. Form based login requires html processing capability so it is not included in this list. Web Services for J2EE product providers must support the following:

- BASIC-AUTH: J2EE servers support basic auth information in the HTTP header that carries the SOAP request. The J2EE server must be able to verify the user ID and password using the authentication mechanism specific to the server. Typically, user ID and password are authenticated against a user registry. To ensure confidentiality of the password information, the user ID and password are sent over an SSL connection (i.e., HTTPS). See the Servlet specification for details on how BASIC-AUTH must be supported by J2EE servers and how a HTTP Digest authentication can be optionally supported. Client container specification of authentication data is described by the J2EE specification section 3.4.4. The EJB and web containers must support deploy time configuration of credential information to use for Web services requests using BASIC-AUTH. The means for this is provider specific though it is typically handled using the generated static stub or dynamic proxy implementation.
- Symmetric HTTPS: J2EE servers currently support authentication through symmetric SSL, when both the requestor and the server can authenticate each other using digital certificates. For the HTTP clients (i.e., SOAP/HTTP), the model is based on the Servlet specification.

9.3.2 Authorization

Web Services for J2EE relies on the authorization support provided by the J2EE containers and is described in the J2EE specification section 3.5.

JAX-RPC Service Endpoint authorization must be defined using the `http-method` element value of `POST`.

9.3.3 Integrity and Confidentiality

A Web Services for J2EE server provider must support HTTPS for hop-by-hop confidentiality and integrity. The WSDL port address may use `https:` to specify the client requirements.

Appendix A. Relationship to other Java Standards

Java APIs for XML

The only required API from this list is JAX-RPC. The rest are listed as being of potential interest. These APIs may become required in a future specification.

JAX-M (JSR 00067) focuses on XML messaging and the Java language.

JAX-R (JSR 00093) defines the Java interfaces to XML registries, like JNDI, ebXML and UDDI. These interfaces provide the mechanism through which client applications find Web services and Web services (and servers) publish their interfaces.

JAX-P (JSR 00005 and 00063) defines APIs for parsing XML

JAX-RPC (JSR 00101) focuses on XML RPC and the Java language, including representing XML based interface definitions in Java, Java definitions in XML based interface definition languages (e.g. SOAP) and marshalling.

XML Trust (JSR00104) defines APIs and protocol for a "Trust Service" to minimize the complexity required for using XML Signatures.

XML Digital Signature (JSR 00105) defines the APIs for XML digital signature services.

XML Digital Encryption (JSR 00106) defines the APIs for encrypting XML fragments.

Java APIs for WSDL (JSR00110) defines the APIs for manipulating WSDL documents.

J2EE APIs

Enterprise JavaBeans 2.0 defines the programming model for implementing Web services which run in the EJB container.

Servlet 2.3 defines the packaging and container service model for implementing Web services which run in the servlet container.

J2EE 1.4, EJB 2.1, and Servlet 2.4 will include the Web services specification defined within this specification. An effort will be made to maintain compatibility and ease migration to J2EE 1.4, but there is no guarantee that compatibility will be ensured. A JSR 109 1.1 maintenance release will address XML Schema based deployment descriptors, the EJB 2.1 Web services view, as well as Holder and Handler class usage in the EJB container for use by J2EE 1.4.

Appendix B. References

JAX-RPC 1.0 Specification. 2002. http://java.sun.com/xml/

JAX-R Specification. 2002. http://java.sun.com/xml/

SOAP 1.1 W3C Note. 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508

WSDL 1.1 W3C Note. 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315

Servlet 2.3 Specification. 2000. http://java.sun.com/j2ee/

J2EE 1.3 Specification. 2001. http://java.sun.com/j2ee/

EJB 2.0 Specification. 2001. http://java.sun.com/j2ee/

Appendix C. Revision History

Appendix C.1. Version 1.0 Final Release

- Updated JAX-RPC mapping DTD to support doc/lit wrapped element.

Appendix C.2. Version 0.95 Final Draft

- Updated license to be the required Specification License Agreement
- Clarified package by reference to be MANIFEST ClassPath use.
- Clarified developer responsibilities for setting the `servlet-mapping` are for the web.xml descriptor. Described deployment tool responsibility for generating one if it doesn't exist
- Clarified container requirements for credential configuration of a service reference.
- Minor editorial changes.

Appendix C.3. Version 0.94

- Clarified binding preference order for container resolution of Port.
- Clarified the Service Interface to be a view of the deployed WSDL the service is bound to.
- JAX-RPC mapping deployment descriptor updated to address void return methods and one-way operations.
- Recommend .xml suffix for mapping deployment descriptor file name.

Appendix C.4. Version 0.93

- Aligned Stub property support with JAX-RPC requirements.
- Clarified port-component to service-impl-bean relationship cardinality is 1-1.
- Clarified requirement for deployment to honor `servlet-mapping` for JAX-RPC Service Endpoint.
- Clarified publishing of deployed WSDL requirements.

Appendix C.5. Version 0.92

- Removed requirement for not providing HandlerChain class.
- Clarified exception thrown to client if Handler inappropriately changes message.
- Clarified use of java:comp/env in Handler methods.
- Clarified use of container services in the web container endpoint.
- DTD DOCTYPEs corrected.
- Editorial cleanup

Appendix C.6. Version 0.8

- Updated JAX-RPC mapping file format

Appendix C.7. Version 0.7

- Completely revised JAX-RPC mapping file to handle missing mapping cases. Support minimal mappings crafted by developed.

Appendix C.8. Version 0.6

- Consolidated client access modes to a modeless Service object. Updated chapter 4 to reflect this and chapter 7 client deployment descriptor.
- Revised platform role responsibilities of chapter 7 for client deployment descriptor to clarify partial WSDL use.
- Added requirements in chapter 6 and 8 for Headers to be defined in the WSDL if they are declared as handled by a Handler.
- Changed the exception thrown if a Handler modifies the request in a way that it shouldn't.
- Clarified use of custom serializers / deserializers as out of scope for this version.

Appendix C.9. Version 0.5

- Added JAX-RPC Mapping deployment descriptor
- Clarified platform role responsibilities
- Clarified deployment
- Terminology changes to sync up with JAX-RPC

Appendix C.10. Version 0.4 Expert Group Draft

- Clarified service development goals.
- Clarified Web services registry goals.
- Clarified container requirements for providing a stub/proxy to the client.
- Changed HandlerRegistry and TypeMappingRegistry access from optional to not supported.
- Clarified use of JAX-RPC Stub properties.
- Added client packaging requirements.
- Strengthened the requirements for exposing an EJB as a Web service.
- Added Handler chapter.