JAKARTA EE

Jakarta Enterprise Web Services

Jakarta Enterprise Web Services Team,
https://projects.eclipse.org/projects/ee4j.jaxws

2.0, July 28, 2020:

Table of Contents

```
Specification: Jakarta Enterprise Web Services

Version: 2.0

Status: Final Release

Release: July 28, 2020
```


Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. [url to this license]"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) 2018 Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Chapter 1. Introduction

This specification defines the Web Services for Jakarta EE (WSEE) architecture. This is a service architecture that leverages the Jakarta EE 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 Jakarta EE developers.

1.1. Target Audience

This specification is intended to be used by:

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

This specification assumes that the reader is familiar with the Jakarta EE platform and specifications. It also assumes that the reader is familiar with Web services, specifically the Jakarta XML Web Services Specification and WSDL documents.

1.2. Acknowledgments

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.

We would also like to acknowledge Roberto Chinnici (JSR-224 Specification Lead), Bill Shannon (JSR-244 Java EE Specification Lead), Rajiv Mordani (JSR-250 Specification Lead), Jitendra Kotamraju, Doug Kohlert, Vivek Pandey, Jerome Dochez, Vijay Ramachandran, Mahesh Kannan and Kenneth Saks (all from Sun Microsystems) for providing invaluable technical input for the maintenance release (version 1.2) of JSR-109 specification.

We would like to thank Roberto Chinnici (Java EE 6 Specification Lead), Bill Shannon (Java EE 6

Specification Lead), Kenneth Saks (EJB 3.1 Specification Lead), Bhakti Mehta and Rama Pulavarthi (all from Sun Microsystems) for providing invaluable input for the 1.3 maintenance release.

1.3. Specification Organization

The next two chapters of this specification outline the requirements and conceptual architecture for Web services support in Jakarta EE environments. Each of the major integration points for Web services in Jakarta EE, 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 attempts to follow the document conventions used by other Jakarta EE specifications.

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.

Chapter 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, SOAP 1.1 and SOAP 1.2.
- Leverage existing Jakarta EE technology.
- Ensure that programmers may implement and package Web services that properly deploy onto application servers that comply with Jakarta EE 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 Jakarta EE.
- Clearly and succinctly define functions that Jakarta EE application server vendors need to provide.
- Define the roles that this specification requires, the functions they perform and their mapping to Jakarta EE platform roles. Define the functions that a Web Services for Jakarta EE product provider must provide to support these roles.
- Support a simple model for defining a new Web service and deploying this into a Jakarta EE application server.

2.1. Client Model Goals

The client programming model should be conformant and compatible with the client programming model defined by Jakarta XML Web Services.

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 Jakarta EE container (e.g. an EJB that uses a Web service), or a Jakarta EE Client Container can call a Web service running in a Web Services for Jakarta EE container.
- Client applications conforming to this specification can call any SOAP 1.1 or SOAP 1.2 based Web service through the HTTP 1.1 or HTTPS SOAP Bindings.
- Programmers using other client environments can call a Web service running in a Web Services for Jakarta EE container. Programmers using languages other than Java must be able to implement SOAP 1.1 or SOAP 1.2 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.

2.2. Service Development Goals

The service development model defines how web service implementations are to be developed and deployed into existing Jakarta EE 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 Jakarta EE or non-Jakarta EE environment.
- Because the Web service implementation must be deployed in a Jakarta EE 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.
- This specification defines the Jakarta EE container based (Web and EJB) runtime such that it is consistent with the programming model defined by the Jakarta XML Web Services specification.
- Support mapping and dispatching SOAP 1.1 or 1.2 requests to methods on Jakarta EE Stateless or Singleton Session Beans.
- Support mapping and dispatching SOAP 1.1 or 1.2 requests to methods on Jakarta XML Web Services Service Endpoint classes in the Web Container.

2.3. Service Deployment Goals

- Web service deployment is declarative. We do this through extending the Jakarta EE model for deployment descriptors and EAR file format. These changes are minimized, however.
- Web service deployment is supported on Jakarta EE 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 Jakarta EE components as a Web service

2.4. 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.

Chapter 3. Overview

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

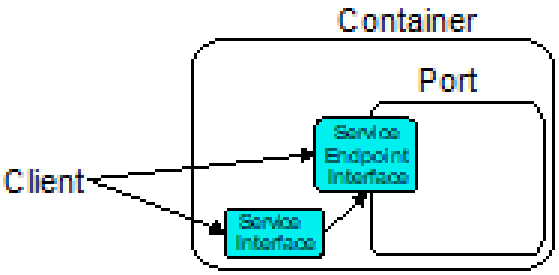
3.2. Web Service

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/Singleton Session EJB or Jakarta XML Web Services 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 Jakarta EE 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.

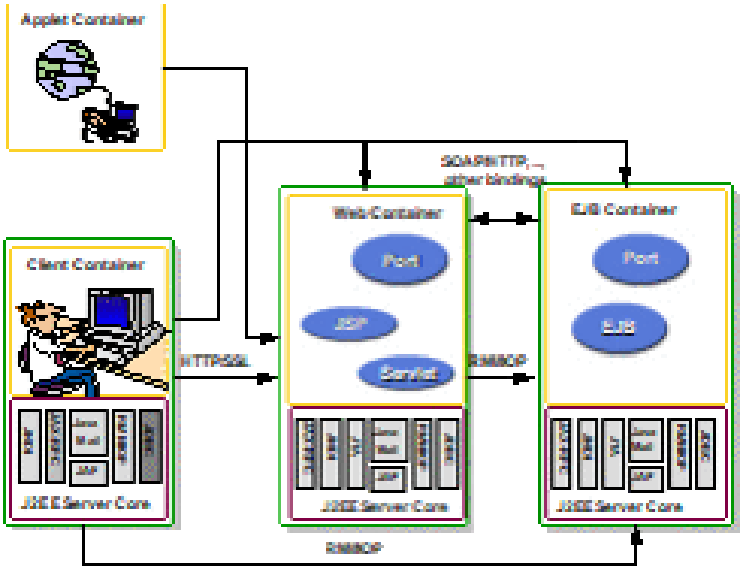
Jakarta XML Web Services defines a programming model mapping of a WSDL document to Java which provides a factory (Service) for selecting which aggregated Port a client wishes to use. See Figure 2 for a logical diagram. In general, the transport, encoding, and address of the Port are transparent to the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



- Figure 2 Client view

3.3. Web Services for Jakarta EE Overview

The Web Services for Jakarta EE 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 Jakarta EE platform include a port component that depends on container functionality provided by the web and EJB containers, and the SOAP/HTTP transport.



- Figure 3 Jakarta EE architecture diagram

Web Services for Jakarta EE 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 Jakarta XML Web Services that may be used to implement Web services, a role based development methodology, portable packaging and Jakarta EE 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 Jakarta EE environment, but does not restrict Web service implementations to just those means. The first is a container based extension of the Jakarta XML Web Services 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 or singleton 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 Jakarta EE environment. This specification does not require that a new container be implemented. Existing Jakarta EE 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 Jakarta XML Web Services 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 Jakarta EE platform roles. There are no new roles defined by this specification. There are two roles specific to Web Services for Jakarta EE used within this specification, but they can be mapped onto existing Jakarta EE platform roles. The Web Services for Jakarta EE product provider role can be mapped to a Jakarta EE product provider role and the Web services container provider role can be mapped to a container provider role within the Jakarta EE specification.

In general, the developer role is responsible for the service definition, implementation, and packaging within a Jakarta EE 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 Jakarta EE 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 Jakarta EE 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 Jakarta EE platform defines a set of standard services a Jakarta EE provider must supply. The Web Services for Jakarta EE specification identifies an additional set of run-time services that are required.

3.6.1. Jakarta XML Web Services

Jakarta XML Web Services is based on the JAX-WS specification from the Java EE specification. This document refers to version 3.0 of the Jakarta XML Web Services specification and APIs unless explicitly noted otherwise.

JAX-WS 2.0 is a follow-on specification to JAX-RPC 1.1. In addition to providing all the run-time services, it improves upon JAX-RPC 1.1 specification by providing support for SOAP 1.2, using JAXB 2.0 specification for all data binding-related tasks, providing support for Web Services metadata etc .

JAX-WS 2.2 adds a complete Web Services addressing support as specified in Web Services Addressing 1.0 - Core, Web Services Addressing 1.0 - Soap Binding, and Web Services Addressing 1.0 - Metadata.

Jakarta XML Web Services 3.0 is a next version of the JAX-WS 2.2 specification. It moves existing APIs

from javax.xml.ws packages to jakarta.xml.ws packages.

3.7. Interoperability

This specification extends the interoperability requirements of the Jakarta EE platform by defining interoperability requirements for products that implement this specification on top of Jakarta EE. 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:

- Jakarta XML-RPC
- Jakarta XML Web Services
- Jakarta Enterprise Edition Specification
- Jakarta Enterprise Beans Specification
- Jakarta Servlet Specification
- WS-I Basic Profile 1.0

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, SOAP 1.2 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 specification 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 specification 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 a Jakarta Enterprise Bean. A client of a Web service can be another Web service, a Jakarta EE component, including a Jakarta EE application client, or an arbitrary Java application. A non-Java application or non-Web Services for Jakarta EE 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 or class
- Service Endpoint interface

The Jakarta XML Web Services Handler interface is considered a container SPI and is therefore not part of the client view.

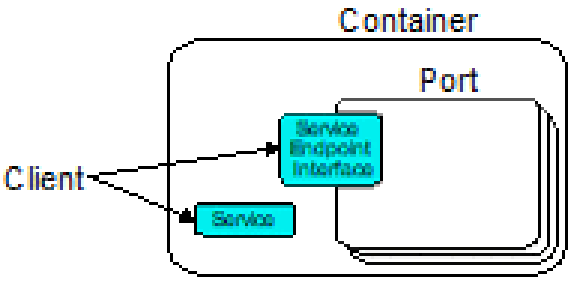


Figure 4 Web Service Client View

The Service Interface/Class 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/Class to obtain access to a Port. The Service interface/class is defined by the Jakarta XML Web Services 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/Class or the Jakarta XML Web Services 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.

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

3.10. Web Service Server View

Chapter 5 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 Jakarta XML Web Services 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 Session Bean: The service provider implements the Web service business logic by creating a stateless session Bean that implements the methods of the Service Endpoint Interface as described in the Jakarta Enterprise Beans specification.
2. A Java class: The service provider implements the Web service business logic according to the requirements defined by the Jakarta XML Web Services Servlet based service implementation model.
3. A Singleton Session Bean: The service provider implements the Jakarta XML Web Services Web service business logic by creating a singleton session bean that implements the methods of the Service Endpoint Interface as described in the Jakarta Enterprise Bean specification.

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 Jakarta XML Web Services specification and Jakarta Enterprise Beans specification for details on the container callback methods.

The container manages the run-time services required by the Web service, such as security. The default behavior requires that if a client accesses a Port with a transaction context, it will be suspended before the Port is accessed. This ensures that remote and local invocations using a SOAP/HTTP binding do not behave differently. Vendors may support transaction propagation (e.g. using WS-AtomicTransaction) as long as the transactional behavior is consistent for local and remote invocations.

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

Packaging of a Web service in a Jakarta EE module is specific to the service implementation methodology, but follows the Jakarta EE requirements for an EJB-JAR file or WAR file. It contains the

Jakarta 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.

3.11. Jakarta EE profiles

The Jakarta EE platform specification defines "profiles" to target specific class of applications.

The Jakarta EE 9 platform removes Jakarta XML-RPC from all profiles, and it includes Jakarta XML Web Services in the full profile.

This specification gives choices for the vendors that want to support only certain containers for Jakarta XML Web Services web services. An Enterprise Web Service implementation must support at least one of the following configurations for Jakarta XML Web Services web services:

- Jakarta XML Web Services web component in a Servlet container
- Stateless or Singleton Session EJB as Jakarta XML Web Services web service

Chapter 4. Client Programming Model

This chapter defines the client programming model of Web Services for Java EE. In general, the client programming model is covered in detail by the Jakarta XML Web Services specification. This specification covers the use of the client programming model in a Jakarta EE environment.

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 Jakarta EE 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 Jakarta EE 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 Jakarta EE clients. It makes no assumption on whether the Web service implementation invoked by the client is hosted by a Web Services for Jakarta EE run-time or some external run-time.

A client uses the Web Services for Jakarta EE run-time to access and invoke the methods of a Web service. A client can be any of the following: Jakarta EE 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 Jakarta XML Web Services 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/Class as defined by the Jakarta XML Web Services 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 Class can be the generic jakarta.xml.ws.Service class or a Generated Service class, which extends jakarta.xml.ws.Service. Further references in this document to the ServiceInterface/Class 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/Class. 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 Jakarta XML Web Services 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/Class in a Web service client deployment descriptor. The client should use the Service interfaces/classes, and not the stubs.

The Service Interface stub/proxy methods provide both service specific (client requires WSDL knowledge) and service agnostic (does not require WSDL knowledge) access to Ports.

A client can use the stub/proxy methods of the Service Interface/Class 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 the portType and bindings.

With Jakarta XML Web Services, a client can also use the Service Class to work at the XML message level using Dispatch APIs. Additionally a client can also make asynchronous invocations using both stubs and Dispatch APIs.

4.2. Specification

The following sections define the requirements for Jakarta EE product providers that implement Web Services for Jakarta EE 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 class, jakarta.xml.ws.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 Jakarta EE product provider is required to provide Service lookup support in the web, EJB, and application client containers.

4.2.2. jakarta.xml.ws.WebServiceRef annotation

With Jakarta XML Web Services, client developer may use the jakarta.xml.ws.WebServiceRef annotation to denote a reference to a Service or a Service endpoint. Lookups using JNDI mechanism for both Service or Service endpoint can also be used under Jakarta XML Web Services. Complete definition of jakarta.xml.ws.WebServiceRef annotation is defined in the Jakarta XML Web Services specification. The containers must ensure that the use of this annotation is supported.

The annotations (for example, *@jakarta.xml.ws.soap.Addressing*) annotated with meta-annotation *jakarta.xml.ws.spi.WebServiceFeatureAnnotation* can be used in conjunction with *@WebServiceRef*. The created reference MUST be configured with annotation's web service feature. If a Jakarta XML Web Services implementation encounters an unsupported or unrecognized feature annotation, an error must be given. Jakarta XML Web Services doesn't define any standard portable web service feature for Service references. But it defines *@Addressing*, *@MTOM*, *@RespectBinding* annotations for SEI proxy references.

By using a web service feature annotation explicitly along with a *@WebServiceRef*, an application overrides WSDL's indication of that feature for the reference. Also, *<enable-mtom>*, *<addressing>*, and *<respect-binding>* deployment descriptor elements can be used to override the *@MTOM*, *@Addressing*, and *@RespectBinding* features respectively for a reference.

The following example illustrates the use of this annotation when declaring a Service:

```
@WebServiceRef(name="java:comp/env/service/AddressBookService")
AddressBookService abf;
```

The same annotation can also be used to declare a SEI proxy reference, the injected SEI proxy reference is configured with MTOM feature:

```
@MTOM
@WebServiceRef(
    name="java:comp/env/service/AddressBookService",
    AddressBookService.class)
AddressBookPort port;

Address address = port.getAddress(“John Doe”);
```

A declared reference can be resolved using lookup functionality specified by Jakarta XML Web Services specification. The following example illustrates the use of this annotation for looking up a Service:

```
@WebServiceRef(lookup="java:comp/env/service/AddressBookService")
AddressBookService other;
```

jakarta.jws.HandlerChain annotation can be used with this annotation to specify handlers on these client side references. More information on the HandlerChain annotation can be found in Jakarta Web Services Metadata specification and also in Chapter 6 of this specification.

If wsdlLocation attribute of WebServiceRef annotation is specified, it is always relative to the root of the module. HTTP URL can also be specified here. The <wsdl-file> element in client deployment descriptor (section 7.2) always overrides the wsdlLocation specified in the annotation. If there is no <wsdl-file> element or wsdlLocation specified in the annotation, then the wsdlLocation attribute of @WebServiceClient annotation on the generated Service class needs to be consulted. (section 7.5 of Jakarta XML Web Services specification).

For co-located clients (where the client and the server are in the same Jakarta EE application unit) with generated Service class, the location of the final WSDL document is resolved by comparing the Service name on the @WebServiceClient annotation on the the generated Service to the Service names of all the deployed port components in the Jakarta EE application unit. This default behavior can be overridden using the <port-component-link> deployment descriptor element. Refer to client deployment descriptor schema Section 7.2.5.

If the name attribute is not specified in this annotation then default naming rules apply as specified in the Jakarta EE specification.

The following table summarizes the relationship between the deployment descriptors for <service-ref>

and member attributes of this annotation.

Deployment Descriptor elements	jakarta.xml.ws.WebServiceRef annotation
<service-ref>	One per @WebServiceRef annotation
<service-ref>/<service-ref-name>	@WebServiceRef.name
<service-ref>/<wsdl-file>	@WebServiceRef.wsdlLocation
<service-ref>/<service-interface>	@WebServiceRef.type when @WebServiceRef.value is not specified. In other words the annotation is used to declare a Service. OR @WebServiceRef.value when @WebServiceRef.type is a Service Endpoint.class The type attribute is implied when this annotation is used on a field. Similar to @Resource annotation in Jakarta Annotations
<service-ref>/<port-component-ref>/<service-endpoint-interface>	@WebServiceRef.type when @WebServiceRef.value is a Service class.
<service-ref>/<port-component-ref>/<port-component-link>	Default mechanism used for co-located case. The deployment descriptor is used only for overriding the default behavior.
<service-ref>/<service-ref-type>	@WebServiceRef.type
<service-ref>/<mapped-name>	@WebServiceRef.mappedName
<service-ref>/<lookup-name>	@WebServiceRef.lookup

- Table 1 Relationship between the deployment descriptor elements and jakarta.xml.ws.WebServiceRef annotation

@WebServiceRef reference instances are not guaranteed to be thread safe. If the instances are accessed by multiple threads, usual synchronization techniques can be used to support multiple

threads.

For declaring multiple references to Web services on a single class jakarta.xml.ws.WebServiceRefs annotation may be used. Complete definition of jakarta.xml.ws.WebServiceRefs annotation is defined in section 7.10 of Jakarta XML Web Services specification. The containers must ensure that the use of this annotation is supported.

4.2.3. Port Lookup

With Jakarta XML Web Services, the client developer can also use JNDI lookups for a Port. This is analogous to using the jakarta.xml.ws.WebServiceRef annotation for Service endpoint. The client side deployment descriptor has been modified to introduce a new optional element <service-ref-type> that declares the type of <service-ref> returned when a dependency injection or JNDI lookup is done. If this element is not specified in the deployment descriptor, then the type of <service-ref> is always a Service class or a generated Service class.

A Jakarta EE product provider is required to provide Port lookup support in the web, EJB, and application client containers.

4.2.4. Service API

The Service API is used by a client to get a stub or dynamic proxy for a Port. A container provider is required to support all methods of the Service interface/class.

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

4.2.4.1. Stub/proxy access

The client may use the following Service class methods to obtain a proxy for a Web service:

```
<T> T getPort(QName portName,
              Class<T> serviceEndpointInterface);
<T> T getPort(java.lang.Class<T> serviceEndpointInterface);
<T> T getPort(Class<T> serviceEndpointInterface,
              WebServiceFeature... features);
<T> T getPort(EndpointReference endpointReference,
              Class<T> serviceEndpointInterface,
              WebServiceFeature... features);
<T> T getPort(QName portName,
              Class<T> serviceEndpointInterface,
              WebServiceFeature... features);
```

The client may also use the additional methods of the Generated Service Interface/Class 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 4.2.5. 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.

4.2.4.2. Service Factory

Two static factory methods Service.create(QName serviceName) and Service.create(URL wsdlLocation, QName serviceName) for creating Service instances rely on specific implementations of ServiceDelegate Class in any Jakarta XML Web Services compliant implementation. The use of these static methods is not recommended in a Web Services for Jakarta EE product. A Web Services for Jakarta EE client must obtain a Service Interface/Class using JNDI lookup as described in section 4.2.1. Container providers are not required to support managed Service instances created using these methods.

4.2.4.3. Service method use with full WSDL

A client developer may use all methods of the Service Interface or class if a full WSDL description is declared in the client deployment descriptor. A mapping file is not required because all of the data binding in Jakarta XML Web Services is done according to the Jakarta XML Binding specification. The port address location attribute of a port using a SOAP/HTTP binding must begin with "http:" or "https:".

If a client developer uses the getPort(SEI) method of a Service Interface/Class 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.

4.2.4.4. Service method use with partial WSDL

A client developer may use the following methods of the Service class:

```
<T> T getPort(java.lang.Class<T> serviceEndpointInterface);
javax.xml.namespace.QName getServiceName();
java.util.Iterator<javax.xml.namespace.QName> getPorts();
java.net.URL getWSDLDocumentLocation();
<T> Dispatch<T> createDispatch(javax.xml.namespace.QName portName,
                               java.lang.Class<T> type,
                               Service.Mode mode);
Dispatch<java.lang.Object> createDispatch(QName portName,
                                          JAXBContext context,
                                          Service.Mode mode);
java.util.concurrent.Executor getExecutor();
void setExecutor(java.util.concurrent.Executor executor);
<T> Dispatch<T> createDispatch(QName portName,
                               Class<T> type,
                               Service.Mode mode,
                               WebServiceFeature... features);
Dispatch<Object> createDispatch(QName portName,
                                JAXBContext context,
                                Service.Mode mode,
                                WebServiceFeature... features);
<T> Dispatch<T> createDispatch(EndpointReference endpointReference,
                               Class<T> type,
                               Service.Mode mode,
                               WebServiceFeature... features);
Dispatch<Object> createDispatch(EndpointReference endpointReference,
                                JAXBContext context,
                                Service.Mode mode,
                                WebServiceFeature... features);
```

A partial WSDL definition is defined as a fully specified WSDL document which contains no service or port elements. A mapping file is not required and ignored if specified, because all of the data binding in Jakarta XML Web Services is done according to the Jakarta XML Binding specification.

Use of other methods of the Service Interface/Class 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.

4.2.4.5. Service method use with no WSDL

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

```
<T> Dispatch<T> createDispatch(javax.xml.namespace.QName portName,
                               java.lang.Class<T> type,
                               Service.Mode mode);
Dispatch<java.lang.Object> createDispatch(QName portName,
                                          JAXBContext context,
                                          Service.Mode mode);
java.util.concurrent.Executor getExecutor();
void setExecutor(java.util.concurrent.Executor executor);
<T> Dispatch<T> createDispatch(QName portName,
                               Class<T> type,
                               Service.Mode mode,
                               WebServiceFeature... features)
Dispatch<Object> createDispatch(QName portName,
                                JAXBContext context,
                                Service.Mode mode,
                                WebServiceFeature... features)
<T> Dispatch<T> createDispatch(EndpointReference endpointReference,
                               Class<T> type,
                               Service.Mode mode,
                               WebServiceFeature... features)
Dispatch<Object> createDispatch(EndpointReference endpointReference,
                                JAXBContext context,
                                Service.Mode mode,
                                WebServiceFeature... features)
```

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

4.2.4.6. Service Interface method behavior

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

Method	Full WSDL	Partial WSDL	No WSDL
void addPort(QName portName, URI bindingId, String endpointAddress)	Normal	Normal	Normal
<T> Dispatch <T> createDispatch(QName portName, Class<T> type, Service.Mode mode)	Normal	Normal	Normal

Method	Full WSDL	Partial WSDL	No WSDL
Dispatch <T> createDispatch(QName portName, JAXBContext context, Service.Mode mode)	Normal	Normal	Normal
Executor getExecutor()	Normal	Normal	Normal
void setExecutor(Executor executor)	Normal	Normal	Normal
HandlerResolver getHandlerResolver()	Normal	Normal	Normal
<T> T getPort(Class<T> SEI)	Normal	Normal	Unspecified
<T> T getPort(QName port, Class<T> SEI)	Normal	Unspecified	Unspecified
Iterator getPorts()	Bound ports	Bound ports	Unspecified
QName getServiceName()	Bound service name	Bound service name	Unspecified
void setHandlerResolver(HandlerResolver handlerResolver)	Normal	Normal	Normal
URL getWSDLDocumentLocation()	Bound WSDL location	Bound WSDL location	Unspecified
Dispatch<Object> createDispatch(EndpointReference epr, Class<T> type, Service.Mode mode, WebServiceFeature… features)	Normal	Normal	Normal

Method	Full WSDL	Partial WSDL	No WSDL
<T> Dispatch<T> createDispatch (EndpointReference epr, JAXBContext context, Service.Mode mode, WebServiceFeature... features)	Normal	Normal	Normal
<T> Dispatch<T> createDispatch(QName portName, java.lang.Class<T> type, Service.Mode mode)	Normal	Normal	Normal
<T> Dispatch<T> createDispatch(QName portName, JAXBContext context, Service.Mode mode, WebServiceFeature ... features)	Normal	Normal	Normal
<T>T getPort(Class<T> SEI, WebServiceFeature ... features)	Normal	Normal	Unspecified
<T>T getPort(EndpointReference epr, Class<T> SEI, WebServiceFeature ... features)	Normal	Unspecified	Unspecified
<T> T getPort(QName portName, Class<T> SEI, WebServiceFeature ... features)	Normal	Unspecified	Unspecified

- Table 2 Service class method behavior with Jakarta XML Web Services

4.2.5. Port Stub and Dynamic Proxy

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

4.2.5.1. 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 or Dynamic Proxy will access a particular Port instance or the same Port instance for multiple invocations.

4.2.5.2. Type narrowing

Proxy classes are not Remote Objects. Hence the use of PortableRemoteObject.narrow(…) is not required.

4.2.6. Jakarta XML Web Services Properties

The Jakarta EE container environment provides a broader set of operational characteristics and constraints for supporting the Stub/proxy properties defined within Jakarta XML Web Services. While support of standard properties for Stub objects is required, their use may not work in all cases in a Jakarta EE environment.

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

- jakarta.xml.ws.security.auth.username
- jakarta.xml.ws.security.auth.password

4.2.6.1. Required properties

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

4.2.7. Jakarta XML Web Services Dispatch APIs

Client developers may use jakarta.xml.ws.Dispatch APIs defined in Jakarta XML Web Services specification. This is a low level API that requires clients to construct messages or message payloads as XML and requires an intimate knowledge of the desired message or payload structure. This is useful in those situations where the client wants to operate at the XML message level.

An instance of jakarta.xml.ws.Dispatch can be obtained by invoking any one of the two createDispatch(…) methods on a Service interface. Details on Dispatch API's and its usage can be referenced at section 4.3 of the Jakarta XML Web Services specification.

4.2.8. Jakarta XML Web Services Asynchronous Operations

Client developer may use asynchronous invocations as defined by the Jakarta XML Web Services

specification. This supports asynchronous invocations through generated asynchronous methods on the Service Endpoint Interface (section 2.3.4 of Jakarta XML Web Services specification) and jakarta.xml.ws.Dispatch (section 4.3.3 of Jakarta XML Web Services specification) interface. There are two forms of asynchronous invocations in Jakarta XML Web Services – Polling and Callback.

4.2.8.1. Polling

Client asynchronous polling invocations must be supported by components running in Servlet container, EJB container and Application Client container, since any of these components can act as Jakarta XML Web Services clients. Client developers can either use the Service Endpoint Interface or jakarta.xml.ws.Dispatch to make asynchronous polling invocations. The usage must meet the requirements defined in section 2.3.4 of Jakarta XML Web Services specification for Service Endpoint Interface or section 4.3.3 of Jakarta XML Web Services specification for jakarta.xml.ws.Dispatch interface.

4.2.8.2. Callback

Client asynchronous callback invocations should only be supported by components running in EJB, Servlet container and Application Client container. Client developers can either use the Service Endpoint Interface or jakarta.xml.ws.Dispatch to implement asynchronous callback invocations. The callback handler must implement jakarta.xml.ws.AsyncHandler interface. The usage should meet the requirements defined in section 2.3.4 of Jakarta XML Web Services specification for Service Endpoint Interface or section 4.3.3 of Jakarta XML Web Services specification for jakarta.xml.ws.Dispatch interface.

It will be the container implementers responsibility to insure that the client developer has access to java:comp/env JNDI context for that component in the callback handler's handleResponse(...) method. The following operations are allowed from within the callback handler:

- JNDI access to java:comp/env
- Resource manager access
- Enterprise bean access

The container implementer is also responsible for ensuring that the context class loader used for the execution of handleResponse() method matches the classloader of the component that made the invokeAsync() call.

Lifecycle of the callback handler is undefined.

It is recommended that the developer use a new instance of the callback handler for each invokeAsync() call to avoid any multi-threading issues.

The behavior of the execution of the callback handler is undefined if the module defining the handler, is undeployed before it is invoked.

Dependency injection is not supported for the callback handler classes. Programmatic JNDI lookups

must be used for getting access to any of the resources.

If no propagated identity is provided for invoking the callback handler, then the handler executes under unauthenticated identity as defined by the container.

The handleResponse() method of the jakarta.xml.ws.AsyncHandler executes in an unspecified transaction context. If the handleResponse() method of the callback handler creates a transaction using the Jakarta Transactions UserTransaction interface then this transaction must be committed or rollbacked before the return of handleResponse() method.

Requirements for asynchronous callback invocations in the EJB container:

- EJB instance cannot be passed as a callback handler instance. User's handler implementation must be a separate class from the Bean class.
- The developer should not attempt to cache the EJBContext of the Bean in the handler. The behavior is undefined if the cached EJBContext is accessed from within the handler.
- The developer should not attempt to cache the Bean instance itself in the Handler. The behavior is undefined if the cached Bean is accessed from within the Handler.

Requirements for asynchronous callback invocations in the Servlet container:

- Servlet instance cannot be passed as a callback handler instance. User's handler implementation is a separate class from the Servlet class.
- The developer should not attempt to cache the Servlet instance itself in the callback handler. The behavior is undefined if the cached Servlet is accessed from within the handler.
- It is recommended that the developer not cache the HttpSession and HttpRequest objects from the Servlet in the callback handler.

4.2.9. JAX-RPC and Jakarta XML Web Services Interoperability

Interoperability between a JAX-RPC client and Jakarta XML Web Services endpoint (or vice-versa) is governed by the requirements defined by the WS-I Basic Profile 1.0. As long as both the client and the server adhere to these requirements, they should be able to interoperate.

4.2.10. MTOM/XOP support

Jakarta XML Web Services compliant implementations are required to support MTOM (Message Transmission Optimization Mechanism)/XOP (XML-binary Optimized Packaging) specifications from W3C. Refer to sections 6.5.2, 7.14.2, and 10.4.1.1 of Jakarta XML Web Services specification. Support for SOAP MTOM/XOP mechanism for optimizing transmission of binary data types is provided by Jakarta XML Binding which is the data binding for Jakarta XML Web Services. Jakarta XML Web Services provides the MIME processing required to enable Jakarta XML Binding to serialize and deserialize MIME based MTOM/XOP packages.

SOAP MTOM/XOP mechanism on the client can be enabled or disabled by any one of the following

ways:

- Programmatically passing MTOMFeature for a Service method that creates a SEI proxy or a Dispatch instance.
- Using <port-component-ref>/<enable-mtom> deployment descriptor element for a corresponding SEI proxy instance.
- Using @MTOM with a @WebServiceRef that creates a SEI proxy instance.

Deployment descriptor mtom elements override the @MTOM annotation for a corresponding SEI instance.

Table : Relationship between deployment descriptor elements and @MTOM

Deployment Descriptor elements	**@MTOM**
<service-ref>/<port-component-ref>/<enable-mtom>	@MTOM.enabled
<service-ref>/<port-component-ref>/<mtom-threshold>	@MTOM.threshold

4.2.11. 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 and a Web services client deployment descriptor (not required if annotations are used) in a Jakarta EE 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 located in the wsdl directory that is co-located with the module deployment descriptor or a subdirectory of it. The developer must not package generated stubs.

Jakarta XML Web Services requires support for a OASIS XML Catalogs 1.1 specification to be used when resolving any Web service document that is part of the description of a Web service, specifically WSDL and XML Schema documents. Refer to section 4.4 of Jakarta XML Web Services specification. The catalog file jax-ws-catalog.xml must be co-located with the module deployment descriptor (WEB-INF/jax-ws-catalog.xml for web modules and META-INF/jax-ws-catalog.xml for the rest).

4.2.12. Web Services Addressing Support

Jakarta XML Web Services clients are required to support Web Services Addressing 1.0 - Core, Web Services Addressing 1.0 - Soap Binding, and Web Services Addressing 1.0 - Metadata.

Web Service Addressing requirements for a client can be specified by any one of the following ways:

- Using <port-component-ref>/<addressing> deployment descriptor element for the corresponding client

- Using @Addressing annotation with the @WebServiceRef of the client
- If the service uses WSDL description, the addressing requirements can be got from the WSDL as per the WS-Addressing 1.0 - Metadata specification.

The above order also defines a precedence order for the addressing requirements. For example, the addressing requirements specified by the @Addressing are overridden by the same from a corresponding <port-component-ref>/<addressing> deployment descriptor element.

Table : Relationship between deployment descriptor elements and @Addressing

Deployment Descriptor elements	**@Addressing**
<service-ref>/<port-component-ref>/<addressing>/<enabled>	@Addressing.enabled
<service-ref>/<port-component-ref>/<addressing>/<required>	@Addressing.required
<service-ref>/<port-component-ref>/<addressing>/<responses>	@Addressing.responses

Jakarta XML Web Services specifies an abstract jakarta.xml.ws.EndpointReference that represents a remote reference to a web service endpoint. jakarta.xml.ws.addressing.W3CEndpointReference class is a concrete EndpointReference implementation for WS-Addressing 1.0 - Core addressing version. Client applications can use an EndpointReference to get a port for an SEI using the getPort methods on jakarta.xml.ws.Service class. Also these EndpointReference objects can appear as SEI method parameters or return type and can be passed across the applications.

A port's EndpointReference can be got using its jakarta.xml.ws.BindingProvider's getEndpointReference method. Occasionally, it is necessary for one application component to create an EndpointReference for another web service endpoint. The W3CEndpointReferenceBuilder class provides a standard API for creating W3CEndpointReference instances for web service endpoints. When creating a W3CEndpointReference for an endpoint published by the same Jakarta EE application, a Jakarta XML Web Services runtime must fill the address (if not set by the application) of the endpoint using its service and port names.

4.2.13. Respect Binding Support

The jakarta.xml.ws.RespectBinding annotation or its corresponding jakarta.xml.ws.RespectBindingFeature web service feature is used to control whether a Jakarta XML Web Services implementation must respect/honor the contents of the wsdl:binding in the WSDL that is associated with the service. See 6.5.3 and 7.14.3 sections in the Jakarta XML Web Services specification.

RespectBinding web service feature on the client can be enabled or disabled by any one of the following ways:

- Programmatically passing RespectBindingFeature for a Service method that creates a SEI proxy or a Dispatch instance.
- Using <port-component-ref>/<respect-binding> deployment descriptor element for a corresponding SEI proxy instance.

- Using @RespectBinding with a @WebServiceRef that creates a SEI proxy instance.

Deployment descriptor <respect-binding> element overrides the @RespectBinding annotation for a corresponding SEI instance.

Table : Relationship between deployment descriptor elements and @RespectBinding

Deployment Descriptor elements	**@RespectBinding**
<service-ref>/<port-component-ref>/<respect-binding>/<enabled>	@RespectBinding.enabled

Chapter 5. Server Programming Model

This chapter defines the server programming model for Web Services for Jakarta EE. 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 Jakarta EE defines two methods of implementing a Web service. It requires the Jakarta XML Web Services 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.

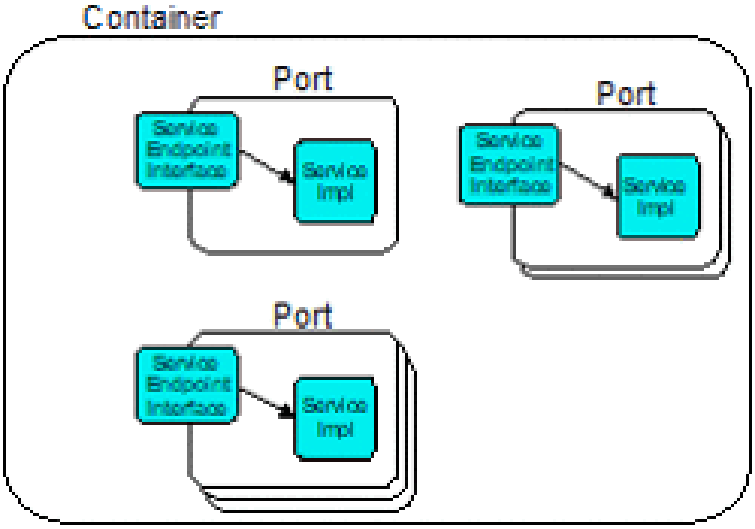
5.1. Goals

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 Jakarta EE 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. Jakarta XML Web Services along with Jakarta Web Services Metadata specification mandates the existence of jakarta.jws.WebService annotated Service Implementation Bean in a Port component. Service Implementation Bean may optionally reference a Service Endpoint Interface but is not required to do so. 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 may implement 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 Jakarta XML Web Services Service Implementation Bean may run in a web container and 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 Jakarta EE component's deployment descriptor. This is discussed further in Chapter 6.3. 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 Jakarta XML Web Services WSDL<→Java mapping rules.

Service Endpoint Interface (SEI) - This interface defines the methods that are implemented by the Service Implementation Bean.

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. When using Jakarta XML Web Services, a developer is not required to have a Web services deployment descriptor . Most of the information in the deployment descriptor is captured in the annotated Service Implementation Bean. A deployment descriptor may be used to override or enhance the information provided in the Service Implementation Bean annotation. 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 Jakarta XML Web Services rules for WSDL<→Java mapping. The SEI is related to the WSDL PortType and WSDL bindings by these rules.

When Jakarta XML Web Services is used, the SEI may be required for client side development only. The Port component developer is not required to provide the SEI or the WSDL document.

5.3.2. Service Implementation Bean

A service implementation bean for a web service can be implemented as follows:

- A Jakarta XML Web Services service endpoint running in a web container
- Stateless Session EJB as a web service
- Singleton Session EJB as a web service

The programming models are fully defined in sections 5.3.2.3 and 5.3.2.4.

A container may use any bean instance to service request.

In a product that also supports Jakarta Contexts and Dependency Injection, an implementation must support use of Jakarta Contexts and Dependency Injection style managed beans as web service classes in an application. Jakarta XML Web Services annotations may be directly applied to these beans. Jakarta Contexts and Dependency Injection specifies the requirements for these container-managed bean instances w.r.t instantiation, injection and other services. Jakarta Contexts and Dependency Injection defines @Dependent pseudo-scope, web service classes must be in that scope. Additionally, a Jakarta XML Web Services service using singleton session EJB can also be in @ApplicationScoped scope. It is an error if the service class has a scope other than the required one.

In a product that also supports Jakarta Managed Beans, an implementation must support use of managed beans as web service classes in an application. Jakarta XML Web Services annotations may be directly applied to managed beans. Jakarta Managed Beans specification specifies the requirements for these container-managed bean instances w.r.t instantiation, injection and other services.

Jakarta XML Web Services along with Jakarta Web Services Metadata specification places additional requirements on Service Implementation Beans detailed in sections 5.3.2.1 and 5.3.2.2.

The developer is only required to provide the jakarta.jws.WebService annotated Service Implementation Bean. The deployment tools could then be used to generate the WSDL document and the SEI using Jakarta XML Web Services rules for Java <→ WSDL mapping.

5.3.2.1. jakarta.jws.WebService annotation

Jakarta XML Web Services along with Jakarta Web Services Metadata specification requires that the Service Implementation Beans must include jakarta.jws.WebService class-level annotation to indicate that it implements a Web Service. Detail requirements and definition of the jakarta.jws.WebService annotation can be found in Jakarta Web Services Metadata specification (section 4.1). If member attributes of the annotation are not specified then server side deployment descriptors (see section 7.1) are used. The member attributes of the annotation can also be overridden by server side deployment descriptors.

A Service Implementation Bean using this annotation is not required to specify a wsdlLocation. If wsdlLocation attribute is specified in the jakarta.jws.WebService annotation, it must follow the packaging rules for the WSDL file detailed in section 5.4. If wsdlLocation attribute is specified, then the WSDL file must exist at that location or can be resolved using the catalog facility specified in section 5.4.4.

The following table shows the relationship between the deployment descriptor elements and this annotation.

Table 1: Relationship between the deployment descriptor elements and jakarta.jws.WebService annotation

Deployment Descriptor element	**jakarta.jws.WebService annotation**
<webservices>/<webservice-description>	One per WSDL document
<webservices>/<webservice-description>/<port-component>	One per @WebService annotation
<webservices>/<webservice-description>/<webservice-description-name>	This is implementation specific
<webservices>/<webservice-description>/<wsdl-file>	@WebService.wsdlLocation

Deployment Descriptor element	**jakarta.jws.WebService annotation**
<webservices>/<webservice-description>/<port-component>/<port-component-name>	@WebService.name (if not specified then its default value as specified in Jakarta Web Services Metadata specification), only if it is unique in the module If the above is not unique then fully qualified name of the Bean class is used to guarantee uniqueness
<webservices>/<webservice-description>/<port-component>/<wsdl-service>	@WebService.serviceName
<webservices>/<webservice-description>/<port-component>/<wsdl-port>	@WebService.portName
<webservices>/<webservice-description>/<port-component>/<service-endpoint-interface>	@WebService.endpointInterface

For Stateless or Singleton Session EJBs using this annotation, the name attribute of the jakarta.ejb.Stateless or jakarta.ejb.Singleton annotation on the Service Implementation Bean class must be used as the <ejb-link> element in the deployment descriptor to map the Port component to the actual EJB. If name attribute in jakarta.ejb.Stateless or jakarta.ejb.Singleton annotation is not specified, then the default value is used as defined in the section 4.4.1 of Jakarta Enterprise Beans specification.

For Servlet based endpoints using this annotation, fully qualified name of the Service Implementation Bean class must be used as the <servlet-link> element in the deployment descriptor to map the Port component to the actual Servlet.

Following default mapping rules apply for Web modules that contain Servlet based endpoints that use this annotation but do not package a web.xml or a partial web.xml:

- fully qualified name of the Service Implementation Bean class maps to <servlet-name> element in web.xml.
- fully qualified name of the Service Implementation Bean class maps to <servlet-class> element in web.xml (also specified in section 7.1.2)
- serviceName attribute of jakarta.jws.WebService annotation prefixed with "/" maps to <url-pattern> element in web.xml. If the serviceName attribute in jakarta.jws.WebService annotation is not specified, then the default value as specified in Jakarta Web Services Metadata specification is used.

The <service-endpoint-interface> element in the deployment descriptor for an implementation bean

must match @WebService.endpointInterface member attribute if it is specified for the bean. Any other value is ignored.

If <wsdl-service> element is provided in the deployment descriptor, then the namespace used in this element overrides the targetNamespace member attribute in this annotation. The namespace in <wsdl-port> element if specified, must match the effective target namespace.

jakarta.jws.WebService annotated Service Implementation Beans can be run either as a Stateless or Singleton Session EJB in an EJB container or as a Jakarta XML Web Services service endpoint in a web container. The two programming models are fully defined in sections 5.3.2.3 and 5.3.2.4.

5.3.2.2. jakarta.xml.ws.Provider interface and jakarta.xml.ws.WebServiceProvider annotation

Service Endpoint Interfaces (SEI) provides a high level Java-centric abstraction that hides the details of converting between Java objects and their XML representations for use in XML-based messages. However, in some cases it is desirable for services to be able to operate at the XML message level. The jakarta.xml.ws.Provider interface in Jakarta XML Web Services offers an alternative to SEIs and may be implemented by Service Implementation Beans wishing to work at the XML message level.

Jakarta XML Web Services requires that these Service Implementation Beans must include jakarta.xml.ws.WebServiceProvider annotation on the class, indicating that it implements the jakarta.xml.ws.Provider interface. Details on the jakarta.xml.ws.WebServiceProvider annotation can be found in Jakarta XML Web Services specification. If member attributes of the annotation are not specified then server side deployment descriptors (see section 7.1) are used. The member attributes of the annotation can also be overridden by server side deployment descriptors .

A WSDL file is required to be packaged with a Provider implementation. If wsdlLocation attribute is specified in the jakarta.xml.ws.WebServiceProvider annotation, it must follow the packaging rules detailed in section 5.4. If wsdlLocation attribute is specified, then the WSDL file must exist at that location or can be resolved using the catalog facility specified in section 5.4.4.

The following table shows the relationship between the deployment descriptor elements and this annotation.

Table 2: Relationship between the deployment descriptor elements and jakarta.xml.ws.WebServiceProvider annotation

Deployment Descriptor element	**jakarta.xml.ws.WebServiceProvider annotation**
<webservices>/<webservice-description>	One per WSDL document
<webservices>/<webservice-description>/<port-component>	One per @WebServiceProvider annotation
<webservices>/<webservice-description>/<webservice-description-name>	This is implementation specific

Deployment Descriptor element	jakarta.xml.ws.WebServiceProvider annotation
<webservices>/<webservice-description>/<wsdl-file>	@WebServiceProvider.wsdlLocation
<webservices>/<webservice-description>/<port-component>/<port-component-name>	Fully qualified name of the Service Implementation Bean is used to guarantee uniqueness
<webservices>/<webservice-description>/<port-component>/<wsdl-service>	@WebServiceProvider.serviceName
<webservices>/<webservice-description>/<port-component>/<wsdl-port>	@WebServiceProvider.portName
<webservices>/<webservice-description>/<port-component>/<service-endpoint-interface>	This deployment descriptor is not required to be specified for Service Implementation Beans that are annotated with @WebServiceProvider

For Stateless or Singleton Session EJBs using this annotation, the name attribute of the jakarta.ejb.Stateless or jakarta.ejb.Singleton annotation on the Service Implementation Bean class must be used as the <ejb-link> element in the deployment descriptor to map the Port component to the actual EJB. If name attribute in jakarta.ejb.Stateless or jakarta.ejb.Singleton annotation is not specified, then the default value is used as defined in the section 4.4.1 of Jakarta Enterprise Beans specification.

For Servlet based endpoints using this annotation, fully qualified name of the Service Implementation Bean class must be used as the <servlet-link> element in the deployment descriptor to map the Port component to the actual Servlet.

Following default mapping rules apply for Web modules that contain Servlet based endpoints that use this annotation but do not package a web.xml or a partial web.xml:

- fully qualified name of the Service Implementation Bean class maps to <servlet-name> element in web.xml.
- fully qualified name of the Service Implementation Bean class maps to <servlet-class> element in web.xml. (also specified in section 7.1.2)
- serviceName attribute of jakarta.xml.ws.WebServiceProvider annotation prefixed with "/" maps to <url-pattern> element in web.xml.

If <wsdl-service> element is provided in the deployment descriptor, then the namespace used in this element overrides the targetNamespace member attribute in this annotation. The namespace in <wsdl-port> element if specified, must match the effective target namespace.

jakarta.xml.ws.WebServiceProvider annotated Service Implementation Beans can be run either as a Stateless or Singleton Session EJB in an EJB container or as a Jakarta XML Web Services service

endpoint in a web container. The two programming models are fully defined in sections 5.3.2.3 and 5.3.2.4.

5.3.2.3. EJB container programming model

A Stateless Session Bean, as defined by the Jakarta Enterprise Beans specification, can be used to implement a Web service to be deployed in the EJB container. A Singleton Session Bean, as defined by the Jakarta Enterprise Beans 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. A Singleton Session Bean is intended to be shared and supports concurrent access. The access rules are specified in the Jakarta Enterprise Beans specification.

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

- The Service Implementation Bean class must be annotated with either jakarta.jws.WebService or jakarta.xml.ws.WebServiceProvider annotation. See section 5.3.2.1 and 5.3.2.2.
- For developers starting from Java, jakarta.jws.WebService annotation on Service Implementation Bean may optionally reference an SEI but is not required to do so. If SEI is not specified, the Service Implementation Bean class implicitly defines a SEI as required by section 3.3 of Jakarta XML Web Services specification. The business methods of the bean must be public and must not be final or static. Only those methods that are annotated with @WebMethod in the Service Implementation Bean, are exposed to the client.
- For developers starting from WSDL, the SEI generated from the WSDL must be annotated with jakarta.jws.WebService annotation. Refer to Jakarta XML Web Services specification. The Service Implementation Bean must be annotated with jakarta.jws.WebService annotation and the endpointInterface member attribute must refer to this generated SEI. 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 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.
- The Service Implementation Bean must have a default public constructor.
- A Service Implementation Bean of a Stateless EJB 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 Service Implementation Bean of Singleton EJB can have a shared state. The singleton session bean instance lives for the duration of the application in which it is created. It maintains its state between client invocations.
- The class must be public, must not be final and must not be abstract.
- The class must not define the finalize() method.

- Currently, it may use jakarta.annotation.PostConstruct or jakarta.annotation.PreDestroy annotation on methods for lifecycle event callbacks. See Jakarta Enterprise Beans specification for more details on this.

jakarta.ejb.Stateless annotation

Currently, a Stateless Session Bean must be annotated with the *jakarta.ejb.Stateless* annotation or denoted in the deployment descriptor as a stateless session bean. The bean class no longer implements the *jakarta.ejb.SessionBean* interface.

The full requirements for Stateless Session Bean are defined in the Jakarta Enterprise Beans specification (EJB Core Contracts and Requirements).

Allowed access to container services

The Jakarta Enterprise Beans specification (EJB Core Contracts and Requirements) defines the allowed container service access requirements.

A stateless or singleton session bean that implements a web service endpoint using the Jakarta XML Web Services APIs should use the jakarta.xml.ws.WebServiceContext, which can be injected by use of the @Resource annotation (see Jakarta Annotations specification), to access message context and security information relative to the request being served. The WebServiceContext interface allows the stateless or singleton session bean instance to get access to the jakarta.xml.ws.handler.MessageContext. Usage of a WebServiceContext must meet the requirements defined by the Jakarta XML Web Services specification.

jakarta.ejb.Singleton annotation

Singleton session bean component, as defined by Jakarta Enterprise Beans specification, provides an easy access to shared state. A Singleton session bean is instantiated once per application. A Singleton session bean must be annotated with the *jakarta.ejb.Singleton* annotation or denoted in the deployment descriptor as a singleton session bean.

The full requirements for Singleton Session Bean are defined by the Jakarta Enterprise Beans specification.

5.3.2.4. Web container programming model

Jakarta XML Web Services Service Endpoint that run within the web container must follow the requirements repeated here.

A Jakarta XML Web Services Service Endpoint can be single or multi-threaded. A Jakarta XML Web Services Service Endpoint must implement jakarta.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. Note, the SingleThreadModel interface has been deprecated in the Servlet 2.4 specification.

The Service Implementation Bean must follow these requirements:

- The Service Implementation Bean class must be annotated with either jakarta.jws.WebService or jakarta.xml.ws.WebServiceProvider annotation. See section 5.3.2.1 and 5.3.2.2.
- For developers starting from Java, jakarta.jws.WebService annotation on Service Implementation Bean may optionally reference an SEI but is not required to do so. If SEI is not specified, the Service Implementation Bean class implicitly defines a SEI as required by Jakarta XML Web Services specification. The business methods of the bean must be public and must not be final or static. Only those methods that are annotated with @WebMethod in the Service Implementation Bean, are exposed to the client.
- For developers starting from WSDL, the SEI generated from the WSDL must be annotated with jakarta.jws.WebService annotation. Refer to Jakarta XML Web Services specification. The Service Implementation Bean must be annotated with jakarta.jws.WebService annotation and the endpointInterface member attribute must refer to this generated SEI. 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 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.
- The Service Implementation Bean must have a default public constructor.
- 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.

The optional @PostConstruct or @PreDestroy annotations

A Service Implementation Bean may use jakarta.annotation.PostConstruct or jakarta.annotation.PreDestroy annotation on methods for lifecycle event callbacks.

The methods annotated with jakarta.annotation.PostConstruct or jakarta.annotation.PreDestroy annotation allow 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 methods that are annotated with jakarta.annotation.PostConstruct or jakarta.annotation.PreDestroy annotations then the container is required to call them in the manner described below.

The container must call the method annotated with jakarta.annotation.PostConstruct before it can start dispatching requests to the methods exposed as Web Service operations of the bean. 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 method annotated with jakarta.annotation.PreDestroy annotation. A container may not call this method while a request is being processed by the bean instance. The

container may not dispatch additional requests to the methods exposed as Web Service operations of the bean after this method is called.

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 Jakarta EE environment (e.g. transactions, JNDI access to the component's environment, etc.) must follow the requirements defined by the Servlet and Jakarta EE specifications.

A Servlet that implements a web service endpoint using the Jakarta XML Web Services APIs should use the jakarta.xml.ws.WebServiceContext, which can be injected by use of the @Resource annotation (see the Jakarta Annotations specification), to access message context and security information relative to the request being served.Usage of a WebServiceContext must meet the requirements defined by the Jakarta XML Web Services specification section 5.3. At runtime, the methods in WebServiceContext serve the same purpose as the methods with the same name defined in jakarta.servlet.http.HttpServletRequest. Service Implementation Beans can get access to HTTPSession and ServletContext using table 9.4 of section 9.4.1.1 of Jakarta XML Web Services specification.

5.3.3. Publishing Endpoints - jakarta.xml.ws.Endpoint

Jakarta XML Web Services provides functionality for creating and publishing Web Service endpoints dynamically using jakarta.xml.ws.Endpoint API. The use of this functionality is considered non-portable in a managed environment. It is required that both the Servlet and the EJB container disallow the publishing of the Endpoint dynamically, by not granting the publishEndpoint security permission. Please refer to details on this in the Jakarta XML Web Services specification.

5.3.4. Service Implementation Bean Life Cycle

The life cycle of a Service Implementation Bean is controlled by the container. The methods called by the container are container/bean specific, but in general are quite similar. The EJB container life cycle can be referenced from Jakarta Enterprise Beans specification.

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 web containers, it calls the method annotated with jakarta.annotation.PostConstruct annotation. For the EJB container, it calls the method annotated with jakarta.annotation.PostConstruct annotation . The jakarta.annotation.PostConstruct callback occurs after any dependency injection has been performed by the container and before the first business method invocation on the bean.

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 method annotated with jakarta.annotation.PreDestroy is called. For the EJB container, the method annotated with jakarta.annotation.PreDestroy is called.

5.3.5. Protocol Binding and jakarta.xml.ws.BindingType annotation

Jakarta XML Web Services specification requires that a developer be able to specify the protocol binding on a Web Service endpoint by using jakarta.xml.ws.BindingType annotation. Jakarta XML Web Services also requires support for the following protocol bindings:

- SOAP1.2 over HTTP - SOAP1.2/HTTP
- SOAP1.1 over HTTP - SOAP1.1/HTTP
- XML over HTTP - XML/HTTP
- SOAP1.1 over HTTP with MTOM enabled
- SOAP1.2 over HTTP with MTOM enabled

Support for overriding the protocol binding specified by BindingType annotation is provided by <protocol-binding> deployment descriptor element for a port component. Refer to section 7.1.2 for details on this deployment descriptor element.

In the event this element is not specified in the deployment descriptors and no BindingType annotation is used, the default binding is used for the endpoint (SOAP1.1/HTTP).

5.3.6. MTOM/XOP support

Jakarta XML Web Services compliant implementations are required to support MTOM (Message Transmission Optimization Mechanism)/XOP (XML-binary Optimized Packaging) specifications from W3C. Refer to Jakarta XML Web Services specification for more information. Support for SOAP MTOM/XOP mechanism for optimizing transmission of binary data types is provided by Jakarta XML Binding which is the data binding for Jakarta XML Web Services. Jakarta XML Web Services provides the MIME processing required to enable Jakarta XML Binding to serialize and deserialize MIME based MTOM/XOP packages.

SOAP MTOM/XOP mechanism on the service can be enabled or disabled by any one of the following ways:

- Using <port-component>/<enable-mtom> deployment descriptor element for a corresponding service

- Using @MTOM with a @WebService that creates a service

Deployment descriptor mtom elements override the @MTOM annotation for a corresponding service. These elements also override if MTOM enabled protocol binding is used. In other words, if MTOM enabled protocol binding is used along with <enable-mtom> set to false, then this feature is disabled. This deployment descriptor must be specified in order to be applied to the protocol binding to enable or disable MTOM. Note that Jakarta XML Web Services recommends the use of MTOM feature instead of mtom enabled bindings: SOAPBinding.SOAP11HTTP_MTOM_BINDING, SOAPBinding.SOAP12HTTP_MTOM_BINDING.

Table 3: Relationship between deployment descriptor elements and @MTOM

Deployment Descriptor element	**@MTOM**
<service>/<port-component>/<enable-mtom>	@MTOM.enabled
<service>/<port-component>/<mtom-threshold>	@MTOM.threshold

5.3.7. Web Services Addressing support

Jakarta XML Web Services services are required to support Web Services Addressing 1.0 - Core, Web Services Addressing 1.0 - Soap Binding, and Web Services Addressing 1.0 - Metadata.

Web Service Addressing requirements for a service can be specified by any one of the following ways:

- Using <port-component>/<addressing> deployment descriptor element for the corresponding service
- Using @Addressing annotation with the service implementation class
- If the service uses WSDL description, the addressing requirements can be specified in the WSDL as per the WS-Addressing 1.0 - Metadata specification.

  The above order also defines a precedence order for the addressing requirements. For example, the addressing requirements specified by the @Addressing are overridden by the same from a corresponding <port-component>/<addressing> deployment descriptor element.

Table 4: Relationship between deployment descriptor elements and @Addressing

Deployment Descriptor element	**@Addressing**
<service>/<port-component>/<addressing>/<enabled>	@Addressing.enabled
<service>/<port-component>/<addressing>/<required>	@Addressing.required
<service>/<port-component>/<addressing>/<responses>	@Addressing.responses

A service's EndpointReference can be got using WebServiceContext's getEndpointReference method during service invocation. Occasionally, it is necessary for one application component to create an EndpointReference for another web service endpoint. The W3CEndpointReferenceBuilder class

provides a standard API for creating W3CEndpointReference instances for web service endpoints. When creating a W3CEndpointReference for an endpoint published by the same Jakarta EE application, a Jakarta XML Web Services runtime must fill the address (if not set by the application) of the endpoint using its service and port names.

5.3.8. RespectBinding support

The jakarta.xml.ws.RespectBinding annotation or its corresponding jakarta.xml.ws.RespectBindingFeature web service feature is used to control whether a Jakarta XML Web Services implementation must respect/honor the contents of the wsdl:binding in the WSDL that is associated with the service. See Jakarta XML Web Services specification for more information.

RespectBinding web service feature on a service can be enabled or disabled by any one of the following ways:

- Using <port-component>/<respect-binding> deployment descriptor element for the corresponding service
- Using @RespectBinding annotation with the service implementation class

  Deployment descriptor <respect-binding> element overrides the @RespectBinding annotation for the corresponding service.

Table 5: Relationship between deployment descriptor elements and @RespectBinding

Deployment Descriptor element	**@RespectBinding**
<service>/<port-component>/<respect-binding>/<enabled>	@RespectBinding.enabled

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 Jakarta XML Web Services Service Endpoint or a Stateless/Singleton session bean as a Jakarta XML Web Services Service Endpoint for the Service Implementation Bean. Port components packaged in a EJB-JAR file must use a Stateless or Singleton Session Bean for the Service Implementation Bean.

The developer is responsible for packaging, either by containment or reference, the WSDL file (not required when annotations are used), Service Endpoint Interface class (optional), Service Implementation Bean class, and their dependent classes, Jakarta XML Web Services generated portable artifacts, along with a Web services deployment descriptor (not required when annotations are used) in a Jakarta EE 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 located in the wsdl directory that is co-located with the module deployment descriptor or a subdirectory of it. Jakarta XML Web Services generated portable artifacts (when starting from Java) include zero or more JavaBean classes to aide in marshaling of method invocations and responses, as well as service-specific

exceptions.

5.4.1. The wsdl directory

The wsdl directory is a well-known location that contains WSDL files and any relative content the WSDL files may reference. WSDL files and their relative references will be published during deployment. See sections 8.2.4 and 8.2.5 for more details.

5.4.2. EJB Module Packaging

Stateless or Singleton 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 Jakarta Enterprise Beans specification. In addition, the Web services deployment descriptor location within the EJB-JAR file is META-INF/webservices.xml. The wsdl directory is located at META-INF/wsdl. See 5.4.3 section for packaging Stateless or Singleton session beans in a WAR file.

5.4.3. Web App Module Packaging

Jakarta XML Web Services Service Endpoints and Stateless/Singleton EJB as Jakarta XML Web Services Service endpoints can be 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. The packaging rules for Stateless or Singleton EJB within a WAR are defined by the EJB specification. In addition, a Web services deployment descriptor is located in a WAR at WEB-INF/webservices.xml and the wsdl directory is located at WEB-INF/wsdl.

5.4.4. Catalog packaging

Jakarta XML Web Services requires support for a OASIS XML Catalogs 1.1 specification to be used when resolving any Web service document that is part of the description of a Web service, specifically WSDL and XML Schema documents. Refer to section 4.4 of Jakarta XML Web Services specification. The catalog file jax-ws-catalog.xml must be co-located with the module deployment descriptor (WEB-INF/jax-ws-catalog.xml for web modules and META-INF/jax-ws-catalog.xml for EJB modules).

5.4.5. Assembly within an EAR file

Assembly of modules containing port components into an EAR file follows the requirements defined by the Jakarta EE 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 or jakarta.ejb.TransactionAttribute annotation.

5.6. Container Provider Responsibilities

In addition to the container requirements described above a container provider must provide a Jakarta XML Web Services runtime.

It is the responsibility of the container provider to support processing Jakarta XML Web Services compliant requests and invoking Ports as described above. The application server must support deployment of these Ports. This specification prescribes the use of the Jakarta XML Web Services Java<→WSDL and Java<→XML Serialization framework for all XML Protocol based Web service bindings. For Jakarta XML Web Services inbound messages, the container will act as the Jakarta XML Web Services 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 Jakarta XML Web Services 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 Jakarta XML Web Services specification.
8. Creating the message envelope appropriate for the transport
9. Sending the message to the originating Web service client.

Chapter 6. Handlers

This chapter defines the programming model for handlers in Web Services for Jakarta EE. Handlers define a means for an application to access the raw 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 section 6.3 for declaration of handlers within deployment descriptors. The Jakarta XML Web Services specification defines the Handler framework. This specification defines Handler use within a Jakarta EE 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.

Jakarta XML Web Services specification defines Logical Handlers and Protocol Handlers. Logical Handlers are protocol agnostic and are unable to affect protocol specific parts of a message. Protocol Handlers operate on message context properties and protocol specific messages. Protocol handlers are specific to a particular protocol and may access and change protocol specific aspects of a message.

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. Jakarta Web Services Metadata specification defines the jakarta.jws.HandlerChain annotation, which can be used with Jakarta XML Web Services in the implementation code to declare HandlerChains associated with a Port component or a Service.

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 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.

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 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 Jakarta EE 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.

Jakarta EE applications that define one or more port components or service references include WSDL descriptions for each of them as well as application logic and (optionally) message handlers associated with them. In order for such applications to behave predictably, all three elements (description, handlers and application logic) must be well aligned. Developers should program handlers carefully in order not to create invalid SOAP envelope format that contradicts WS-I BP 1.0 requirements or violates the message schema declared in the WSDL. In particular, containers cannot provide any guarantees beyond those specified as part of the interoperability requirements on the behavior of an application that violates the assumptions embedded in a WSDL document either in its business logic or in message handlers.

6.2. Specification

This section defines the requirements for Jakarta XML Web Services Handlers running Jakarta Enterprise Web Services. Differences between this specification and the Jakarta XML Web Services 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 Jakarta EE provider is required to provide all interfaces and classes of the jakarta.xml.ws.handler package.

A Jakarta XML Web Services for Jakarta EE provider is required to provide an implementation of

HandlerResolver that returns the Handler Chain with Handlers specified in the deployment descriptor or the jakarta.jws.HandlerChain annotation. The ordering of the Handlers in the HandlerChain must follow the requirements specified in Jakarta XML Web Services specification section 9.2.1.2. In addition to this, the ordering of any given type of Handler (logical or protocol) in the deployment descriptor must be maintained.

A Web Services for Jakarta EE provider is required to provide an implementation of MessageContext.

A Web Services for Jakarta EE provider is required to provide all the interfaces of the jakarta.xml.ws.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 5.3.2.3 and 5.3.2.4. The concurrency of a Jakarta XML Web Services 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.

In a product that also supports Jakarta Contexts and Dependency Injection, an implementation must support use of CDI-style managed beans as Jakarta XML Web Services handler classes in an application. Jakarta Contexts and Dependency Injection specifies the requirements for these container-managed bean instances w.r.t instantiation, injection and other services. Jakarta Contexts and Dependency Injection specification defines @Dependent pseudo-scope, handler classes must be specified in that scope. It is an error if the handler class has a scope other than the required one.

In a product that also supports Jakarta Managed Beans, an implementation must support use of managed beans as Jakarta XML Web Services handler classes in an application. Jakarta Managed Beans specification specifies the requirements for these container-managed bean instances w.r.t instantiation, injection and other services.

6.2.2.1. Handler Life Cycle with Jakarta XML Web Services

The Jakarta XML Web Services based container manages the lifecycle of handlers by invoking any methods of the handler class annotated as lifecycle methods before and after dispatching requests to the handler itself.

The lifecycle of a handler instance begins when the container creates a new instance of the handler class.

- The handler instance is initialized by invoking the lifecycle method annotated with jakarta.annotation.PostConstruct (defined by Jakarta Annotations specification). This method is only called after all injections requested by the Handler are completed. Once the handler instance is created and initialized it is placed into the Ready state. While in the Ready state the Jakarta XML Web Services runtime may invoke other handler methods as required.

- The lifecycle of a handler instance ends when the container stops using the handler for processing inbound or outbound messages and invokes the lifecycle method annotated with jakarta.annotation.PreDestroy.

The lifecycle methods annotated with jakarta.annotation.PostConstruct and jakarta.annotation.PreDestroy in the Handler implementation allows the container to notify a Handler instance of impending changes in its state. Detailed requirements for Handler lifecycle are described in section 9.3.1 of Jakarta XML Web Services specification. The Handler may use the notification to prepare its internal state for the transition. The container is required to call lifecycle methods annotated with jakarta.annotation.PostConstruct and jakarta.annotation.PreDestroy as described below.

The container must carry out any injections (if any) requested by the handler, typically via the @Resource annotation (see Jakarta Annotations specification). After all the injections have been carried out, the container must invoke the method carrying a jakarta.annotation.PostConstruct annotation. This method must have a void return type and take zero arguments. The handler instance is then ready for use and other handler methods may be invoked.

The container must call the lifecycle method annotated with jakarta.annotation.PreDestroy annotation on any Handler instances which it instantiated, before releasing a handler instance from its working set. A container must not call this method while a request is being processed by the Handler instance. The container must not dispatch additional requests to the Handler after the this method is called.

The requirements for processing any RuntimeException or ProtocolException thrown from handle<action>() method of the handler are defined in sections 9.3.2.1 and 9.3.2.2 of the Jakarta XML Web Services specification.

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 handle<action>() methods in a Jakarta XML Web Services based container. It is not required that the same Handler instance service handleMessage() or handleFault() method invocation of any given request in the Jakarta XML Web Services based container.

6.2.2.2. jakarta.jws.HandlerChain annotation

The jakarta.jws.HandlerChain annotation from Jakarta Web Services Metadata specification (imported by Jakarta XML Web Services) may be declared on Web Service endpoints (those declared with the jakarta.jws.WebService or jakarta.xml.ws.WebServiceProvider annotation) or on Web Service references (those declared with the jakarta.xml.ws.WebServiceRef annotation). This annotation is used to specify the handler chain to be applied on the declared port component or Service reference. Details on the jakarta.jws.HandlerChain annotation can be found in section 4.6 of Jakarta Web Services Metadata specification. If this annotation is used, the handler chain file for this must be packaged with

the application unit according to the packaging rules in Section 6.3.

The deployment descriptors on port component or Service reference override the jakarta.jws.HandlerChain annotation specified in the implementation.

The <handler-chains> element in the deployment descriptor is used for specifying the handlers on a port component or Service reference. This deployment descriptor allows for specifying multiple handler chains such that all handlers in a handler chain could be specific to a Service name, a Port name or a list of protocol bindings. Patterns on Service names and Port names are also allowed, where in the handlers in a handler chain could be specific to a Service name pattern or Port name pattern. Refer to Chapter 7 for details on the deployment schema for handlers.

Jakarta XML Web Services based container provider is required to support this annotation. They are also required to provide an implementation of HandlerResolver that returns a handler chain with handlers specified in the deployment descriptor or the jakarta.jws.HandlerChain annotation. The ordering of the handlers in the handler chain must follow the requirements specified in section 9.2.1.2 of Jakarta XML Web Services specification. In addition to this, the ordering of any given type of Handler (logical or protocol) in the deployment descriptor must be maintained.

6.2.2.3. 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 Jakarta XML Web Services 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 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 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 EE.6.2.3 of the Jakarta EE specification for more details.

6.2.2.4. Transactions

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

Handlers must not demarcate transactions using the jakarta.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 jakarta.xml.ws.handler.Handler interface or one of its subinterfaces.

With Jakarta XML Web Services, the handler allows for resources to be injected, typically by using the @Resource annotation. So a Handler.handle<action>() method may access the component's context and environment entries by using any resources that were injected. It can also use JNDI lookup of the "java:comp/env" context and accessing the env-entry-names defined in the deployment descriptor by performing a JNDI lookup. See chapter 15 of the Jakarta Enterprise Beans specification - *Jakarta Enterprise Beans Core Contracts and Requirements* 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. The element init-params in the deployment descriptors is no longer used for Jakarta XML Web Services based container. If needed, the developer should use the environment entry elements (<env-entry>) declared in the application component's deployment descriptor for this purpose. These can be injected into the handler using the @Resource annotation or looked up using JNDI.

A Handler implementation that implements the jakarta.xml.ws.handler.soap.SOAPHandler interface must contain all the headers information needed by it. Additionally, in this case the soap-header element declared in the deployment descriptor is not required since that information is embedded in the implementation of the Handler class.

Only a Handler implementation that implements the jakarta.xml.ws.handler.soap.SOAPHandler interface must implement the 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 Jakarta XML Web Services 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-entrys of the component it is associated with by using JNDI to lookup an appropriate subcontext of java:comp/env. It may also access these if they are injected using the @Resource annotation. Access to the java:comp/env contexts must be supported from the method annotated with jakarta.annotation.PostConstruct and handle<action>() methods. Access may not be supported within the method annotated with jakarta.annotation.PreDestroy annotation.

A Handler may access transactional resources defined by a component's resource-refs. Resources are accessed under a transaction context according to section 6.2.2.5.

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.

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

In a Jakarta XML Web Services based container, a Handler chain is processed according to the Jakarta XML Web Services specification section 9.2.1.2. In addition to this, the ordering of any given type of Handler (logical or protocol) in the deployment descriptor or in the handler configuration file specified in the jakarta.jws.HandlerChain annotation, must be maintained.

The container must ensure that for EJB based Web Service endpoints with both Handlers and EJB Interceptors present, the Handlers must be invoked before any EJB business method interceptor methods.

A Jakarta XML Web Services based container must carry out any injections (if any) requested by the handler, typically via the @Resource annotation (see section 2.2 of Jakarta Annotations specification). A Jakarta XML Web Services handler should use the jakarta.xml.ws.WebServiceContext, which is an injectable resource, to access message context and security information relative to the request being served. A unique Handler instance must be provided for each Port component declared in the deployment descriptor or annotated by jakarta.jws.WebService or jakarta.xml.ws.WebServiceProvider annotations.

The container provider must ensure that for a Jakarta XML Web Services based EJB endpoint executing in the EJB container, with both Handlers and EJB Interceptors present, the java.util.Map<String,Object> instance returned by invoking WebServiceContext.getMessageContext() method in the Jakarta XML Web Services Handler, is the same Map instance that is obtained by invoking InvocationContext.getContextData() in the EJB Interceptor. This common Map instance would allow for sharing of data (if required) between the Handlers and Interceptors.

A Jakarta XML Web Services based container must call the lifecycle method annotated with jakarta.annotation.PostConstruct within the context of a Port component's environment. The container must ensure the Port component's env-entrys are setup for this lifecycle 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 Jakarta XML Web Services SOAPHandler 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.

6.3. Packaging

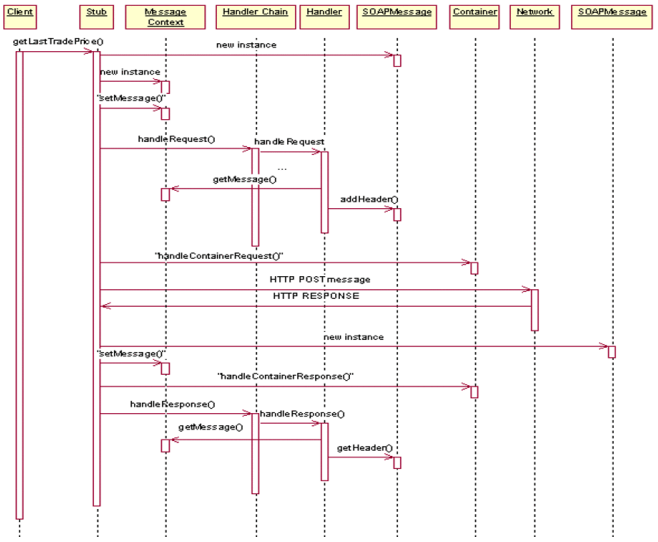
A developer is required to package, either by containment or reference, the Handler class and its dependent classes in the module with the deployment descriptor information that references the Handler classes. A developer is responsible for defining the handler chain information in the deployment descriptor.

The handler chain file in the jakarta.jws.HandlerChain annotation is required to be packaged in the module. It must follow the requirements for location as specified in Jakarta Web Services Metadata specification. Additionally, the handler chain file can also be packaged and specified in the annotation such that, it is accessible as a resource from the ClassPath. At runtime, container providers must first try to access the handler chain file as per the locations specified in Jakarta Web Services Metadata specification specification. Failing that, they must try to access it as a resource from the ClassPath. If more than one resources are returned from the ClassPath, then the first one is used.

6.4. Object Interaction Diagrams

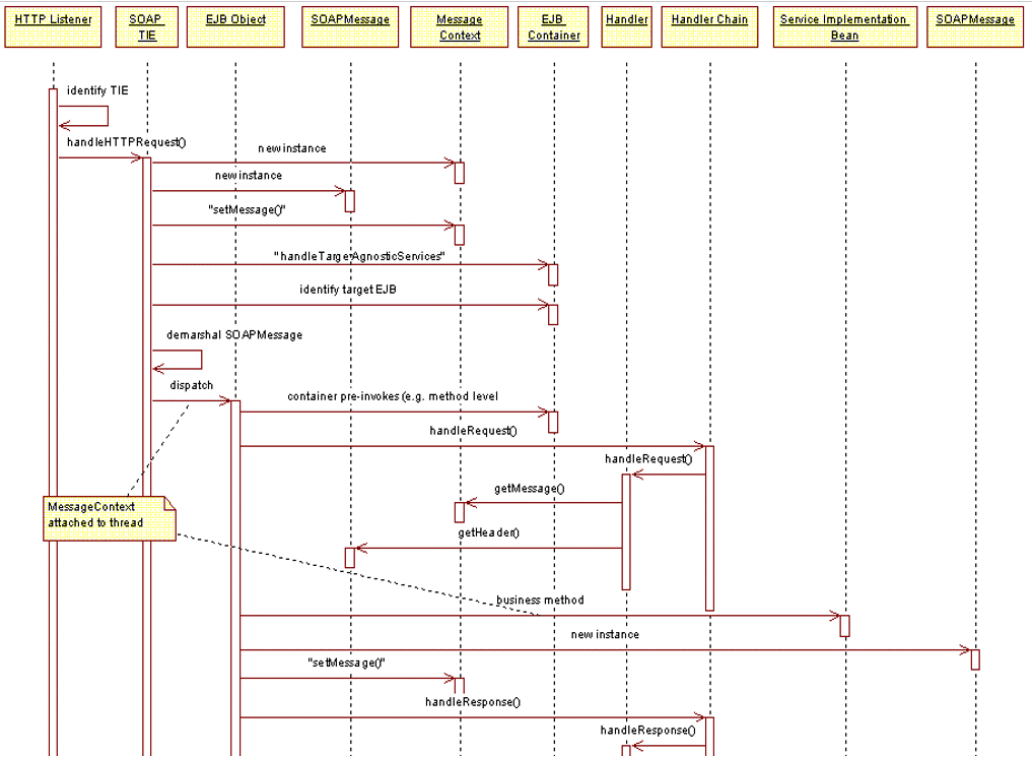
This section contains object interaction diagrams for handler processing. In general, the interaction diagrams are meant to be illustrative.

6.4.1. Client Web service method access

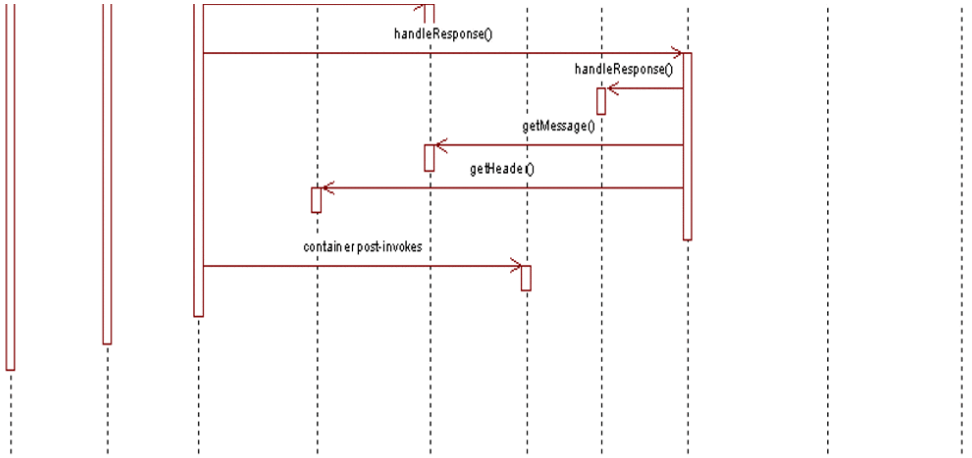


- Figure 8 Client method invoke handler OID

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

Chapter 7. Deployment Descriptors

This chapter describes the various deployment descriptors used for Web Services for Jakarta EE 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 Jakarta EE enabled container. With Jakarta XML Web Services the use of webservices.xml is optional since the annotations can be used to specify most of the information specified in this deployment descriptor file. The deployment descriptors are only used to override or augment the annotation member attributes. The packaging of the webservices.xml deployment descriptor file is defined in sections 5.4.2 and 5.4.3. 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 EJB annotations or module specific deployment descriptor and Web Services annotations or the webservices.xml deployment descriptor. Service Implementations using a stateless or singleton session bean may use EJB annotations with no deployment descriptor file. If the EJB annotations are not specified then the stateless or singleton session bean must be defined in the ejb-jar.xml deployment descriptor file using the session element. Servlet based web service endpoints using Jakarta XML Web Services are not required to provide the web.xml deployment descriptor file (see section 5.3.2.1). See the Jakarta Enterprise Beans 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 Jakarta XML Web Services Service Endpoint, the servlet-link element associates the port-component with Jakarta XML Web Services Service Endpoint class defined in the web.xml by the servlet-class element. For a stateless or singleton 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. If the Service Implementation is a stateless session EJB, the developer must also specify the Service Endpoint Interface in the EJB deployment descriptor using the service-endpoint element. See the Jakarta Enterprise Beans specification for more details.
- **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. The WSDL file may reference (e.g. import) other files contained within the module using relative references. It may also reference other files external to the module using an explicit URL. Relative imports are declared relative to the file defining the import. Imported files may import other files as well using relative locations or explicit URLs. It is recommended that the WSDL file and relative referenced files be packaged in the wsdl directory as described in section 5.4.1. Relative references must not start with a "/".
- **Service QName.** In addition to specifying the WSDL document, the developer may also specify the WSDL Service QName in the wsdl-service element for each Port defined in the deployment descriptor when Jakarta XML Web Services based runtime is used. This element is required if the port-component being defined is a Provider Interface defined by Jakarta XML Web Services.
- **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.
- **MTOM/XOP support.** The developer may specify if MTOM/XOP support for the port-component is enabled or disabled by using enable-mtom element when Jakarta XML Web Services based runtime is used.
- **Addressing support.** The developer may specify an addressing support for the port-component by using addressing element when Jakarta XML Web Services based runtime is used.
- **RespectBinding support.** The developer may specify a respect binding support for the port-component by using respect-binding element when Jakarta XML Web Services based runtime is used.
- **Protocol Binding.** The developer may override the protocol binding specified by BindingType annotation by specifying the URI or a pre-defined token (like ##SOAP11_HTTP, ##SOAP12_HTTP, ##XML_HTTP etc.) in the protocol-binding element when Jakarta XML Web Services based runtime is used. The default protocol binding is ##SOAP11_HTTP. If this element is not specified then the

default value is assumed. The pre-defined tokens essentially act as alias for the actual URI for the protocol binding. The URIs for these pre-defined tokens are listed below:

- ##SOAP11_HTTP - "http://schemas.xmlsoap.org/wsdl/soap/http"
- ##SOAP12_HTTP - "http://www.w3.org/2003/05/soap/bindings/HTTP/"
- ##SOAP11_HTTP_MTOM - "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true"
- ##SOAP12_HTTP_MTOM - "http://www.w3.org/2003/05/soap/bindings/HTTP/?mtom=true"
- ##XML_HTTP - "http://www.w3.org/2004/08/wsdl/http"

- **Handlers.** A developer may optionally specify handlers associated with the port-component using handler-chains element when Jakarta XML Web Services based runtime is used. With Jakarta XML Web Services, the soap-header element declared in the deployment descriptor is not required since that information is embedded in the implementation of the Handler class. With Jakarta XML Web Services, init-params element in the deployment descriptors is no longer used. If needed, the developer should use the environment entry elements (<env-entry>) declared in the application component's deployment descriptor for this purpose. These can be injected into the handler using the @Resource annotation or looked up using JNDI.
- **Servlet Mapping.** A developer may optionally specify a servlet-mapping, in the web.xml deployment descriptor. 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 Jakarta EE are an extension of the assembler responsibilities as defined by the Jakarta Enterprise Beans, Servlet, and Jakarta EE specifications. The assembler creates a deployable artifact by composing multiple modules, resolving cross-module dependencies, providing annotation overrides 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-role elements, or may add new handler elements or new handler-chain elements. With Jakarta XML Web Services, Handler implementation must contain all the soap-header information needed by it.

See also the assembler responsibilities defined in section 7.2.3.

7.1.4. Deployer responsibilities

The deployer responsibilities are defined by the Jakarta EE, Jakarta Enterprise Beans, 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 XML Schema

The XML Schema for the Web service deployment descriptor is described at https://jakarta.ee/xml/ns/jakartaee/jakartaee_web_services_2_0.xsd

7.2. Service Reference Deployment Descriptor Information

This section defines the function of the Service Reference XML schema file, its use within modules, the platform roles and responsibilities for defining instance data, and the format.

7.2.1. Overview

The Service Reference XML schema defines the schema for service reference entries. These entries declare references to Web services used by a Jakarta EE component in the web, EJB, or application client container. With Jakarta XML Web Services, these entries are not required if jakarta.xml.ws.WebServiceRef annotation is used. If the Web services client is a Jakarta EE 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 module's deployment descriptor file.

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 Jakarta XML Web Services Service Interface/Class class returned by the JNDI lookup.
- **Service Reference type:** The service-ref-type element declares the type of the service-ref element that is injected or returned when a JNDI lookup is done. This must be either a fully qualified name of Service class or the fully qualified name of Service endpoint interface class. This is an optional

element.

- **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.
- **MTOM/XOP support.** The developer may specify if MTOM/XOP support for the port-component-ref is enabled or disabled by using enable-mtom element when Jakarta XML Web Services based runtime is used.
- **Addressing support.** The developer may specify an addressing support for the port-component-ref by using *addressing* element when Jakarta XML Web Services based runtime is used.
- **RespectBinding support.** The developer may specify a respect binding support for the *port-component-ref* by using *respect-binding* element when Jakarta XML Web Services based runtime is used.

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 4.2.4.4 or 4.2.4.5 are used. The WSDL file may reference (e.g. import) other files contained within the module using relative references. It may also reference other files external to the module using an explicit URL. Relative imports are declared relative to the file defining the import. Imported files may import other files as well using relative locations or explicit URLs. Relative references must not start with a "/".
- **Service Port.** If the specified wsdl-file has more than one service element, the developer must specify the service-qname.
- **Handlers.** The developer may optionally use the handler-chains element when specifying handler chains associated with the service-ref under Jakarta XML Web Services based runtime.

7.2.3. Assembler responsibilities

In addition to the responsibilities defined within the Jakarta EE 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 Jakarta EE 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 service-ref element of the module's 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-role elements, or may add new handler elements or new handler-chain elements. With Jakarta XML Web Services, Handler implementation must contain all the soap-header information needed by it.
- **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 Jakarta EE 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 Service Reference XML Schema

This section defines the XML Schema for the service-ref at https://jakarta.ee/xml/ns/jakartaee/jakartaee_web_services_client_2_0.xsd. This schema is imported into the common Jakarta EE schema and is used by the application client, web, and EJB module deployment descriptor schemas to declare service-refs. See the Jakarta EE 7 and corresponding versions of Servlet and EJB specifications for more details on specifying a service-ref in the deployment descriptors.

Chapter 8. Deployment

This chapter defines the deployment process requirements and responsibilities. Deployment tasks are handled by the Jakarta EE deployer platform role using tools typically provided by the Web Services for Jakarta EE 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 Jakarta EE specification.

8.1. Overview

This section describes an illustrative process of deployment for Web Services for Jakarta EE. 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 Jakarta EE artifacts and the second phase is standard Jakarta EE 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 Web service metadata annotations or 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.
- Jakarta XML Web Services service components are only packaged within a WAR file.
- Stateless or Singleton session bean Web services are only packaged within an EJB-JAR or WAR file.
- The WSDL bindings used by the WSDL ports are supported by the Web Services for Jakarta EE 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 Jakarta XML Web Services Service Endpoint requires different handling than deployment of a session bean service.

If the implementation is a Jakarta XML Web Services Service Endpoint, a servlet is generated to handle

parsing the incoming SOAP request and dispatch it to an instance of the Jakarta XML Web Services service component. The generated servlet class is dependent on threading model of the Jakarta XML Web Services Service Endpoint. The web.xml deployment descriptor is updated to replace the Jakarta XML Web Services Service Endpoint class with the generated servlet class. If the Jakarta XML Web Services 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 url-pattern of the servlet-mapping. If the implementation is a stateless or singleton 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, the servlet obtains a reference to an instance of an appropriate EJBObject and dispatches the request to the stateless or singleton session 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 referenced by Web service metadata annotations or 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 annotated with jakarta.xml.ws.WebServiceRef or 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 WebServiceRef annotation or the client deployment descriptor (the Service interface class, the Service Endpoint Interface class, and WSDL ports the client wants to access). In general the procedure includes providing an implementation of the Jakarta XML Web Services Service interface/class 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 that uses either the jakarta.xml.ws.WebServiceRef annotation or is 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 Jakarta EE deployable artifact, the deployment process continues using normal deployment processes.

It is recommended that containers provide logging functionality similar to that of the WS-I "Monitor" tool. Such containers would log all incoming and outgoing messages in the format defined by the WS-I Testing Tools group and would allow capturing SOAP messages exchanged over the HTTPS protocol in a way that allows analysis by the WS-I tools.

8.1.1. Jakarta XML Web Services HTTP SPI

Jakarta XML Web Services includes HTTP SPI that allows a deployment to use any available Jakarta XML Web Services runtime for HTTP transport. This allows 109 implementations to use the Jakarta XML Web Services runtime in a Jakarta SE platform when it is available. For more details on the HTTP SPI, see the section 6.6 of Jakarta XML Web Services specification.

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.

A deployment tool must be able to deploy a WS-I Basic Profile 1.0 compliant application. Validating an application for WS-I Basic Profile 1.0 conformance is considered a value add.

8.2.2. Generate Web Service Implementation classes

Generation of any run-time classes the container requires to support a Jakarta XML Web Services Service Endpoint or Stateless or Singleton Session Bean Service Implementation is provider specific. The behavior of the run-time classes must match the information provided by annotations or deployment descriptor settings of the component. A Jakarta XML Web Services 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 jakarta.ejb.Stateless annotation or <session> element and the <assembly-descriptor> in the ejb-jar.xml deployment descriptor. A Singleton Bean Service Implementation must match the behavior defined by the jakarta.ejb.Singleton annotation or <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 wsdlLocation element in the Web service annotations (described in section 5.3.2.1 and 5.3.2.2) or 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 container must not update a WSDL file located in the document root of a WAR file.

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 service-ref WSDL

The deployment tool and/or container must make the WSDL document that a service-ref (or a jakarta.xml.ws.WebServiceRef annotated Web service reference) is bound to available via a URL returned by the Service Interface getWSDLDocumentLocation() method. This may or may not be the same WSDL document packaged in the module. The process of publishing the bound service-ref (or a jakarta.xml.ws.WebServiceRef annotated Web service reference) WSDL is analogous to publishing deployed WSDL, but only the service-ref (or a jakarta.xml.ws.WebServiceRef annotated Web service reference) that is bound to it is required to have access to it. A Web Services for Jakarta EE provider is required to provide a URL that maintains the referential integrity of the WSDL document the service-ref (or a jakarta.xml.ws.WebServiceRef annotated Web service reference) is bound to if the wsdl-file (wsdlLocation in WebServiceRef) element refers to a document located in the wsdl directory or one of its subdirectories.

8.2.5. 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.wsdl and a port-component which declares a port QName within the WSDL document. When deployed, the port address in StockQuoteDescription.wsdl 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.

A Web Services for Jakarta EE provider is required to support publishing a deployed WSDL document if the wsdlLocation element in the Web service annotations (described in section 5.3.2.1 and 5.3.2.2) or Web services deployment descriptor (webservices.xml) wsdl-file element refers to a WSDL file

contained in the wsdl directory or subdirectory, as described in section 5.4.1. A vendor may support publication of WSDL files packaged in other locations, but these are considered non-portable. A provider may publish the static content (e.g. no JSPs or Servlets) of the entire wsdl directory and all its subdirectories if the deploy tool cannot compute the minimal set of documents to publish in order to maintain referential integrity. The recommended practice is to place WSDL files referenced by a wsdlLocation element in the Web service annotations or wsdl-file element and their relative imported documents under the wsdl directory.

Web Services for Jakarta EE providers are free to organize the published WSDL documents however they see fit so long as referential integrity is maintained. For example, the wsdl directory tree may be collapsed to a flat published directory structure (updating import statements appropriately). Clients should not depend on the wsdl directory structure being maintained during publication. Access to relatively imported documents should only be attempted by traversing the published WSDL document at the location chosen by the deployer.

Requirements for publishing WSDL documents to a UDDI V2 directory are described by the WS-I Basic Profile 1.0 specification.

8.2.6. Service and Generated Service Interface/Class implementation

The container must provide an implementation of the Jakarta XML Web Services Service Interface/Class. There is no requirement for a Service Implementation to be created during deployment. The container may substitute a Generated Service Interface/Class Implementation for a generic Service Interface/Class Implementation.

The container must provide an implementation of the Jakarta XML Web Services Generated Service Interface/Class if the Web services client deployment descriptor defines one. A Generated Service Interface/Class Implementation will typically be provided during deployment.

The Service Interface/Class 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.

8.2.7. 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.

Jakarta XML Web Services specification makes no distinction between stubs and dynamic proxies, but talks only about proxies and they must conform to Jakarta XML Web Services specification section 4.2.3.

The container is required to support credential propagation as defined in section 4.2.6 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.

8.2.8. 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.

8.2.9. Deployment failure conditions

Deployment may fail if:

- The webservices.xml deployment descriptor is invalid or Web service metadata annotations specified are incorrect
- 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.
- Jakarta XML Web Services service components are not packaged within a WAR file.
- Stateless or Singleton session bean Web services are not packaged within an EJB-JAR or WAR file.
- The WSDL bindings used by the WSDL ports are not supported by the Web Services for Jakarta EE 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 module's deployment descriptor containing:

```
<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>
```

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.

Chapter 9. Security

This section defines the security requirements for Web Services for Jakarta EE. 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 Jakarta EE 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 Jakarta EE 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

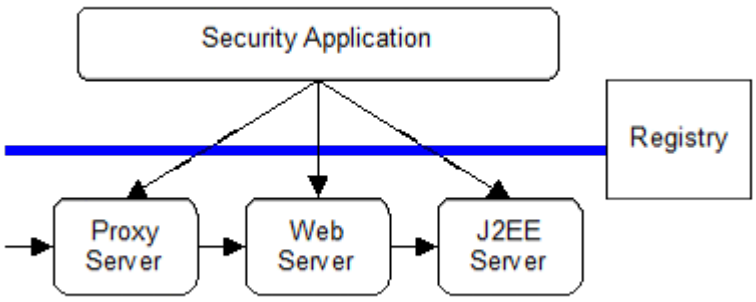
Since the Web services architecture builds on existing component technologies, intra-enterprise authentication is no different than today's approaches. In order for two or more parties to communicate securely they may need to exchange security credentials. Web service's security is used to exchange many 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 Jakarta EE 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 Jakarta EE Server receives the request. In these cases, a Proxy Server or Web Server forwards authentication credentials into the Jakarta EE Application Server. The Jakarta EE application server handles the request similar to how it handles other HTTP

requests.

Two forms of authentication are available for use within Web Services for Jakarta EE based on existing Jakarta EE functionality. These are HTTP BASIC-AUTH and Symmetric HTTP, which are defined by the Jakarta 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 embedded third party model.

In addition to Jakarta EE 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). Jakarta Enterprise Web Services specification 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 Jakarta EE 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 Jakarta EE Servers based on the Jakarta EE security model to only allow authorized access to the methods of EJBs and Servlets/JSPs. Authorization to Web services implemented as Jakarta XML Web Services 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 Jakarta EE 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 Jakarta EE 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

Jakarta EE 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 Jakarta EE servers implement support for audit records, but defers to the Jakarta EE 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 Jakarta EE 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 Jakarta EE 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 Jakarta EE 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 Jakarta EE Application Server.

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

The server relies on HTTPS for hop-by-hop confidentiality and integrity .

9.3. Specification

The following sections define the requirements for implementing security for Web Services for Jakarta

EE.

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 Jakarta EE product providers must support the following:

- BASIC-AUTH: Jakarta EE servers support basic auth information in the HTTP header that carries the SOAP request. The Jakarta EE 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 Jakarta EE servers and how a HTTP Digest authentication can be optionally supported. Client container specification of authentication data is described by the Jakarta EE 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. Also, these containers must provide a way to configure each instance of the generated static stub or dynamic proxy implementation with credential information. The means for this is provider specific though it is typically handled using the generated static stub or dynamic proxy implementation.
- Symmetric HTTPS: Jakarta EE 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 Jakarta EE relies on the authorization support provided by the Jakarta EE containers and is described in the Jakarta EE specification section 3.5.

Jakarta XML Web Services Service Endpoint authorization must be defined using the http-method element value of POST.

9.3.3. Integrity and Confidentiality

A Web Services for Jakarta EE 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

Jakarta APIs for XML

The only required APIs from this list are Jakarta XML Web Services, Jakarta Web Services Metadata and Jakarta Annotations. The rest are listed as being of potential interest. These APIs may become required in a future specification.

Jakarta XML Web Services is a follow-on to Jakarta XML-RPC and extends it.

Jakarta Web Services Metadata defines Web Services Metadata using annotations to simplify the programming model for Web Services

Jakarta Annotations defines Common Annotations for the Java Platform

Jakarta Enterprise Beans defines the programming model for implementing Web services which run in the EJB container.

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

Jakarta XML-RPC 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.

Jakarta XML Registries 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.

Java APIs for XML

These APIs are listed as being of potential interest. They may become required in a future specification.

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

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

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.

Appendix B: References

1. Jakarta XML Web Services 3.0 https://jakarta.ee/specifications/xml-web-services/3.0/
2. Jakarta XML Binding 3.0 https://jakarta.ee/specifications/xml-binding/3.0/
3. Jakarta Web Services Metadata https://jakarta.ee/specifications/web-services-metadata/3.0/
4. Jakarta Annotations 2.0 https://jakarta.ee/specifications/annotations/2.0/
5. Jakarta XML-RPC 1.1 https://jakarta.ee/specifications/xml-rpc/1.1/
6. Jakarta XML Registries 1.0 https://jakarta.ee/specifications/xml-registries/1.0/
7. SOAP 1.1 W3C Note, 2000 http://www.w3.org/TR/2000/NOTE-SOAP-20000508
8. SOAP 1.2 W3C specification, 2003 http://www.w3.org/TR/soap12/
9. WSDL 1.1 W3C Note, 2001 http://www.w3.org/TR/2001/NOTE-wsdl-20010315
10. MTOM/XOP W3C Recommendation, 2005 http://www.w3.org/TR/soap12-mtom/
11. Jakarta Servlet 5.0 https://jakarta.ee/specifications/servlet/5.0/
12. Jakarta EE 9 https://jakarta.ee/specifications/platform/9/
13. Jakarta Enterprise Beans 4.0 https://jakarta.ee/specifications/enterprise-beans/4.0/
14. Jakarta Context Dependency Injection 3.0 https://jakarta.ee/specifications/cdi/3.0/
15. Web Services Addressing 1.0 - Core. W3C Recommendation. 2006. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509/
16. Web Services Addressing 1.0 – Soap Binding. W3C Recommendation. 2006. http://www.w3.org/TR/2006/REC-ws-addr-soap-20060509/
17. Web Services Addressing 1.0 - Metadata. W3C Recommendation. 2007. http://www.w3.org/TR/2007/REC-ws-addr-metadata-20070904

Appendix C: Revision History

Version 2.0

- Changed specification version and license.
- Changed package name to jakarta.
- Removed references to Jakarta XML-RPC.
- Updated the XML Schema for the Web service deployment descriptor to accommodate changed namespace (Section 7.1.5).
- Updated the XML Schema for the service-ref accommodate changed namespace (Section 7.2.5).

Version 1.3

- Added Singleton Session EJB requirements (Sections 2.1.2, 3.2, 3.3.1, 3.10, 3.11, 4.2.14, 5.3.2.1, 5.3.2.2, 5.3.2.3, 5.3.2.3.2, 5.3.2.3.3, 5.4, 5.4.2, 5.4.3, 7.1.2, 8.1, 8.2.2, 8.2.10)
- Added addressing feature support (Sections 3.6.2, 4.2.2, 4.2.13, 5.3.8, 7.1.2, 7.2.2)
- Added mtom feature support (Sections 4.2.2, 4.2.10, 5.3.7)
- Added respect binding feature support (Sections 4.2.14, 5.3.9)
- Updated @WebServiceRef with features, and lookup functionality (Section 4.2.2)
- Added Java EE profiles (Section 3.11)
- Updated with new JAX-WS Service methods (Sections 4.2.4.1, 4.2.4.5, 4.2.4.6, 4.2.4.7)
- Added JSR-299 and Managed Beans requirements for service implementation bean (Section 5.3.2)
- Updated packaging requirements (Sections 5.4, 5.4.2, 5.4.3)
- Added JSR-299 and Managed Beans requirements for handler classes (Section 6.2.2)
- Updated inline schemas with the links (Sections 7.1.5, 7.2.5, 7.3.5)

Version 1.2 Proposed Draft 2

- Clarified transaction propagation (Section 3.10)
- Clarified that the use of @WebServiceRef not required with JAX-WS (Section 4.2.2)
- Mapping for <service-ref-type> client descriptor specified (Section 4.2.2)
- Clarification on usage of @WebServiceRefs was added (Section 4.2.2)
- Added section on JNDI lookups for Ports (Section 4.2.3)
- Clarification on JAX-RPC mapping file (Section 4.2.4.5)
- Added support for JAX-WS Asynchronous callback operations from EJBs (Section 4.2.8.2)
- Added a section clarifying interoperability between JAX-RPC and JAX-WS clients (Section 4.2.9)
- Added support for enabling/disabling MTOM/XOP mechanism in the client (Section 4.2.10)

- Added clarification in Packaging section (4.2.12)
- Clarified uniqueness requirement for @WebService.name in Java EE module (Section 5.3.2.1)
- Clarified mapping for <ejb-link> and <servlet-link>(Section 5.3.2.1, Section 5.3.2.2)
- Clarified mapping rules for Servlet endpoints with no web.xml (Section 5.3.2.1, Section 5.3.2.2)
- Clarified mapping between <service-endpoint-interface> and @WebService.endpointInterface (Section 5.3.2.1)
- Clarified mapping relationship between <port-component> and @WebService (Section 5.3.2.1)
- Clarified mapping relationship between <port-component> and @WebServiceProvider (Section 5.3.2.2)
- Clarified that a WSDL file must be packaged with a Provider implementation (Section 5.3.2.2)
- Clarified the usage of WebServiceContext with Stateless Session bean (Section 5.3.2.3.2)
- Clarified the usage of WebServiceContext with Servlets (Section 5.3.2.4.2.2)
- Disallowed the publishing of Endpoints (Section 5.3.3)
- Added support for new BindingTypes from JAX-WS (Section 5.3.6)
- Corrected the MTOM deployment descriptor element name and clarified its usage (Section 5.3.7)
- Clarifications in Packaging (Section 5.4)
- Clarifications in packaging regarding mixing of JAX-RPC and JAX-WS components in a module (Section 5.4)
- Added clarification on use of TransactionAttribute annotation (Section 5.5)
- Clarified order of invocation for Handlers and EJB interceptors (Section 6.2.4)
- Clarification for alignment between Handlers and EJB interceptors for EJB endpoints (Section 6.2.4)
- Clarified that deployment descriptors were optional (Section 7.1.1 and 7.1.2)
- Added new protocol binding tokens (Section 7.1.2)
- Changes in section on Web Services Deployment Descriptor XML Schema (Section 7.1.5)
- Clarified that deployment descriptors were optional (Section 7.2.1)
- Added <service-ref-type> deployment descriptor element (Section 7.2.2)
- Added <enable-mtom> deployment descriptor element (Section 7.2.2)
- Changes in section on Web Services Client Service Reference XML Schema (Section 7.2.5)
- Clarified that JAX-WS specification makes no distinction between stubs and proxies (Section 8.2.7)
- Relevant references to deployment descriptor elements or file webservices.xml were fixed to include Web services metadata annotations allowed by JAX-WS
- Restriction on use of Mandatory transaction attribute was removed (Section 8.1 and 8.2.10)

Version 1.2 Proposed Draft

- Relevant occurrences of J2EE in the entire specification were changed to Java EE
- Relevant occurrences of JAX-RPC in the entire specification were changed to add JAX-WS to it.
- Relevant occurrences of SOAP 1.1 in the entire specification were changed to add SOAP 1.2 to it.
- Added support for client side @WebServiceRef annotation to access Web Service (Section 4.2.2)
- Changes to support javax.xml.ws.Service Class in JAX-WS (Section 4.2.3)
- Changes for JAX-WS properties (Section 4.2.5)
- Use of javax.xml.ws.Dispatch APIs (Section 4.2.6)
- Support for JAX-WS Asynchronous operations (Section 4.2.7)
- Support for OASIS XML Catalogs specification (Section 4.2.9)
- Clarification on use of Service Endpoint Interface with JAX-WS (Section 5.3.1)
- Added support for @WebService annotation on Service Implementation Bean (Section 5.3.2.1)
- Added support for @WebServiceProvider annotation on Service Implementation Bean (Section 5.3.2.2)
- Changes in EJB container programming model related to JAX-WS (Section 5.2.3.3)
- Web container programming model for JAX-WS (Section 5.3.2.4.2)
- Added support for specifying protocol binding (Section 5.3.5)
- Added support for enabling/disabling MTOM/XOP (Section 5.3.6)
- Added support for Catalog packaging (Section 5.4.4)
- Changes in JAX-WS Handler Programming Model (Section 6.2.2)
- Added new section on Handler Lifecycle with JAX-WS (Section 6.2.2.2)
- Added new section on @HandlerChain annotation (Section 6.2.2.3)
- Clarification in section on Security (Section 6.2.2.4)
- Changes in section on Developer Responsibilities related to JAX-WS (Section 6.2.3)
- Changes in section on Container Provider Responsibilities related to JAX-WS (Section 6.2.4)
- Changes in section on Packaging related to handlerchain file (Section 6.3)
- Added description of new deployment descriptor elements like <wsdl-service>, <enable-mtom>, <protocol-binding>, <handler-chains> (Section 7.1.2)
- Removed the old schema and added new updated server side schema (Section 7.1.5)
- Removed the old schema and added new updated client side schema (Section 7.2.5)
- Clarified that the JAX-RPC mapping file is not required in JAX-WS (Section 7.3)

Version 1.1 Final Release

- Clarified anonymous type qname-scope use

- Clarified parsing of anonymous type qnames
- Clarified portable anonymous type array forms
- Added missing mapping in anonymous type mapping example

Version 1.1 Proposed Final Draft

- Updated XML schemas.
- Holder and Handler support are now required for the EJB container.
- Corrected port address requirements.
- Clarified handler access of resources.
- Clarified mappings for xsd:any and anonymous types.
- Updated to support WS-I Basic Profile 1.0. Clarified interoperability requirements.

Version 1.1 Public Draft 3

- Removed section on exposing an existing EJB.
- Clarified WSDL packaging and publishing requirements when dealing with relative imports.

Version 1.1 Public Draft

- Removed J2EE 1.3 deployment requirements. Appendix B added describing optional support for J2EE 1.3 based deployment.
- Replaced DTD deployment descriptors with XML schema deployment descriptors.

Version 1.0 Final Release

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

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.

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.

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.

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

Version 0.8

- Updated JAX-RPC mapping file format

Version 0.7

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

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.

Version 0.5

- Added JAX-RPC Mapping deployment descriptor

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

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.