JAKARTA EE

Jakarta Web Services Metadata

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

Table of Contents

```
Specification: Jakarta Web Services Metadata

Version: 3.0

Status: Final Release

Release: July 17, 2020
```

Copyright (c) 2019, 2020 Eclipse Foundation.

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.

Scope

Jakarta Web Services Metadata defines a programming model for Web Services in Java, use of metadata, a non-normative processing model for metadata annotated web service source files, runtime requirements for a container, and annotations used for WSDL, binding, and configuration.

Chapter 1. Introduction

This specification defines a simplified programming model that facilitates and accelerates the development of enterprise Web Services. Jakarta EE standard deployment technologies, APIs, and protocols require the Jakarta EE developer to master a substantial amount of information. This specification reduces the amount of information required to implement Web Services on Jakarta EE by using metadata to specify declaratively the Web Services that each application provides. The metadata annotates the Java source file that implements the Web Service. Although the metadata is human-readable and editable with a simple text editor, graphical development tools can represent and edit the Java source file with higher levels of abstraction specific to Web Services. These tools represent a simpler and more powerful development environment than do traditional coding tools that are used to develop source code with low level APIs.

This specification relies on the JSR-175 specification "A Program Annotation Facility for the Java™ Programming Language" for the Web Services metadata that annotates a Web Service implementation. This document uses JSR-175 features as described in the Public Draft Specification of JSR-175.

Jakarta Web Services Metadata Specification defines the syntax and semantics of Java Web Service (JWS) metadata and default values and implementers are expected to provide tools that map the annotated Java classes onto a specific runtime environment. This specification does not define a Java environment in which Web Services are run; however, the use of a Java SE 8 compiler is assumed. In particular, it is assumed in Jakarta Web Services Metadata that features such as Jakarta XML Web Services and Jakarta Enterprise Web Services, along with the compiler and language extensions from JSR-175, are present.

A Jakarta Web Services Metadata implementation MUST produce a deployable JWS application that can run in the target Java environment. The deployed application MUST exhibit the proper behavior described by the Web Services metadata and Java source code. Any two Jakarta Web Services Metadata processors starting from the same valid annotated JWS file MUST produce equivalent Web Service applications, even though they may deploy in very different Java environments. This consistency ensures portability of Jakarta Web Services Metadata compliant Java files.

1.1. Expert Group Members

The following people have been part of the original Expert Group

Alexander Aptus (Togethersoft Corporation)

John Bossons

Charles Campbell

Shih-Chang Chen (Oracle)

Alan Davies (SeeBeyond Technology Corp)

Stuart Edmondston (BEA Systems)

John Harby

RajivMordani (Sun Microsystems)

Michael Morton (IBM)

Simon Nash (IBM)

Mark Pollack

Srividya Rajagopalan (Nokia)

Krishna Sankar (Cisco Systems)

Manfred Schneider (SAP AG)

John Schneider (BEA Systems)

Kalyan Seshu (Paramati Technologies)

Rahul Sharma (Motorola)

Michael Shenfield (Research In Motion)

Evan Simeone (PalmSource)

Brian Zotter (BEA Systems)

1.2. Acknowledgements

Manoj Cheenath (BEA Systems), Don Ferguson (BEA Systems), Chris Fry (BEA Systems), Neal Yin (BEA Systems), Beverley Talbott (BEA Systems), Matt Mihic, Jim Trezzo, Doug Kohlert (Sun Microsystems), Jitendra Kotamraju (Sun Microsystems), and Rama Pulavarthi (Sun Microsystems) have all provided valuable technical input to this specification.

1.3. Conventions

The keywords '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[11].

1.4. Objectives

The following objectives describe the scope of this specification:

- Define an annotated Java syntax for programming Web Service applications.

- Provide a simplified model for Web Service development that facilitates and accelerates development.

- Provide a syntax that is amenable to manipulation by tools.

- Define a standard for to building and deploying Web Services without requiring knowledge and implementation of generalized APIs and deployment descriptors.

This specification addresses the need to simplify:

- Development of server applications that conform both to basic SOAP and WSDL standards.

- Building Web Services that can be deployed with the core Web Services APIs and existing Jakarta EE standards.

- Separate control of public Web Service message contracts and private implementation signatures, because in practice public and private formats evolve on different schedules.

It is not a goal of this specification to support every feature or to enable the creation of every Web Service that it is possible to write within existing specifications (Jakarta XML Web Services [5] provides finer control over the resultant Web Service). The goal is to make it easy to build the most common types of Web Services.

Chapter 2. Concepts

This section summarizes the following concepts and processes of the Jakarta Web Services Metadata specification:

- Programming model for Jakarta Web Services Metadata annotated Web services

- Use of metadata in Jakarta Web Services

- Non-normative processing model for a JWS file

- Runtime requirements for a Jakarta Web Services Metadata container

- Annotations used for WSDL, binding and configuration

The metadata is formally described in section 4.

2.1. Programming Model Overview

Jakarta Web Services Metadata Specification, along with Jakarta XML Web Services Specification and Jakarta Enterprise Web Services Specification, defines a programming model for building a Web Service. A developer who builds a Web Service with these technologies is required to write and manage several artifacts: a WSDL document describing the external Web Service contract; a service endpoint interface defining the Java representation of the Web Service interface; a service implementation bean containing the Web Service implementation; and one or more deployment descriptors linking the WSDL, interface, and implementation into a single artifact. Jakarta Web Services Metadata Specification simplifies this model by allowing the developer to write only the service implementation bean - *actual business logic* – and use annotations to generate the remaining artifacts.

2.2. Development Models

Jakarta Web Services Metadata Specification defines several different models of Web Service development. Only the Start with Java development model is REQUIRED by implementations.

2.2.1. Start with Java

Following the "Start with Java" development model, the developer begins by writing a Java class to expose as a Web Service. The developer then runs this Java class through the Jakarta Web Services Metadata processor, which produces WSDL, schema, and other deployment artifacts from the annotated Java code. By default, the WSDL produced from the Java source follows the Java to XML/WSDL mapping defined by Jakarta XML Web Services Specification. However, the developer may customize the generated WSDL through annotations on the Java source. For example, the developer may use the @WebService.name annotation to set explicitly the name of the wsdl:portType representing the Web Service.

Jakarta Web Services Metadata Specification also supports a development model where the service is

defined in Java but the messages and types are defined in XML schema. In this model, the developer starts by defining a set of types and elements in XML schema. The schema definitions are passed through a "schema to Java" compiler to produce a corresponding set of Java types. The resulting Java types are then used as parameters and return values on methods in an annotated service implementation bean. The WSDL produced from this service implementation bean imports or directly includes the schema definitions that match the Java types used by the service.

2.2.2. Start with WSDL

Following the "start with WSDL" development model, the developer uses Jakarta Web Services Metadata Specification to implement a predefined WSDL interface. Typically, this process begins with the developer passing a pre-existing WSDL 1.1 file through an implementation-supplied tool to produce a service endpoint interface that represents the Java contract, along with Java classes that represent the schema definitions and message parts contained in the WSDL. The developer then writes a service implementation bean that implements the service endpoint interface. In this model, Jakarta Web Services Metadata annotations supply implementation details that are left out of the original WSDL contract, such as binding or service location information.

2.2.3. Start with WSDL and Java

Following the "start with WSDL and Java" development model, the developer uses Jakarta Web Services Metadata annotations to associate a service implementation bean with an existing WSDL contract. In this model, the Jakarta Web Services Metadata annotations map constructs on the Java class or interface to constructs on the WSDL contract. For example, the developer could use the @WebMethod.operationName annotation to associate a method on the service implementation bean with a predefined wsdl:operation. A Jakarta Web Services Metadata implementation that supports this model MUST provide feedback when a service implementation bean no longer adheres to the contract defined by the original WSDL. The form that this feedback takes depends on the implementation. For example, a source editing tool might provide feedback by highlighting the offending annotations, while a command line tool might generate warnings or fail to process a service implementation bean that does not match the associated WSDL.

2.3. Processor Responsibilities

The term "Jakarta Web Services Metadata processor" denotes the code that processes the annotations in a Jakarta Web Services Metadata JWS file to create a runnable Web Service. Typically this involves generating the WSDL and schemas that represent the service and its messages and the deployment descriptors that configure the service for the target runtime. It may also result in the generation of additional source artifacts.

This specification does not require implementations to follow a particular processing model. An implementation MAY use whatever processing model is appropriate to its environment, as long as it produces a running Web Service with the proper contract and runtime behavior. For example, one implementation might process the Jakarta Web Services Metadata annotations directly within the Java compiler to generate a deployable Web Service as the output of compilation; another might provide

tools to convert a compiled service implementation bean into a set of artifacts that can be deployed into the container; and a third might configure its runtime container directly off the Java source or class file. Each implementation is conformant with Jakarta Web Services Metadata Specification as long as it produces a Web Service with the proper runtime behavior.

2.4. Runtime Responsibilities

The runtime environment provides lifecycle management, concurrency management, transport services, and security services. This specification defines the set of annotations that a developer may use to specify declaratively the behavior of an application, but does not define a specific runtime environment or container. Instead, the Jakarta Web Services Metadata processor is responsible for mapping the annotated Java classes onto a specific runtime environment. This specification envisions – but does not require – several such runtime environments:

- Automatic deployment to a server directory – This is a "drag and drop" deployment model, similar to that used by Jakarta Server Pages. The annotated JWS file is copied in source or class form to a directory monitored by the container. The container examines the annotations in the file to build a WSDL and configures the runtime machinery required for dispatching. This approach provides a simplified deployment model for prototyping and rapid application development (RAD).

- Automatic deployment with external overrides – Similar to previous approach, but with the addition of an external configuration file containing overrides to annotations. The additional configuration file allows an administrator to customize the behavior or configuration of the Web Service – such as the endpoint URL - without changing the Java source.

- Generation of Jakarta EE Web Services - In this model, a tool uses the metadata in the annotated Java class to generate a Jakarta EE Web Service based on Jakarta Enterprise Web Services and Jakarta XML Web Services Specifications. The initial Web Service is generated from the annotated Java source, and the result can be further customized through standard deployment tools, including Jakarta Deployment plans. This feature allows customization of externally modifiable properties at deployment or runtime, without requiring access to the source file for modification and recompilation.

2.5. Metadata Use

The metadata that annotates the service implementation bean conforms to the JSR-175 specification and the specific Jakarta Web Services Metadata *annotation type* declarations that are defined in this specification in conjunction with the JSR-175 metadata facility. These *annotation type* declarations are contained in packages that MUST be imported by every Jakarta Web Services Metadata JWS source file. JSR-175 provides the syntax for expressing the annotation element declarations that are in these packages. This specification specifies the contents of the jakarta.jws and jakarta.jws.soap packages (see attached APIs).

Developers use a standard Java compiler with support for JSR-175 to compile and validate the service implementation bean. The compiler uses the annotation type declarations in the jakarta.jws and jakarta.jws.soap packages to check for syntax and type mismatch errors in the Web Service metadata.

The result of compilation is a Java .class file containing the Web Service metadata along with the compiled Java code. The class file format for these annotations is specified by JSR-175. Any Web Service metadata that this JSR designates as runtime-visible is also accessible through the standard java.lang.reflect classes from the run-time environment.

2.5.1. Error Checking

Although the compiler can check for syntax and type errors by using the annotation type declaration, syntactically valid metadata may still contain semantic errors. Implementations MUST provide a validation mechanism to perform additional semantic checking to ensure that a service implementation bean is correct. The validation MAY be performed in a separate tool or as part of deployment.

Examples of semantic checks include:

- Ensuring that annotation values match extended types. The Java compiler can ensure that a particular annotation member-value is of the type specified in the annotation type declaration. However, JSR-175 restricts annotations to simple types such as primitives, Strings, and enums. As a result, the compiler cannot ensure that, for example, an annotation member is a valid URL. It can only verify that the member is a String. The Jakarta Web Services Metadata implementation MUST perform the additional type checking to ensure that the value is a valid URL.

- Ensuring that annotations match the code. For example, the developer MAY use the @Oneway annotation to indicate that a particular operation does not produce an output message. If the operation is marked @Oneway, it MUST NOT have a return value or out/in-out parameters. The Jakarta Web Services Metadata implementation MUST provide feedback if this constraint is violated.

- Ensuring that annotations are consistent with respect to other annotations. For example, it is not legal to annotate a method with the @Oneway annotation unless there is also a corresponding @WebMethod annotation. The Jakarta Web Services Metadata implementation MUST ensure these constraints are met.

Note: Certain types of errors MAY only be caught when the Web Service is deployed or run.

2.5.2. Default Values

Jakarta Web Services Metadata Specification defines appropriate defaults for most annotation members. This feature exempts the JWS author from providing tags for the most common Web Service definitions. Although this specification uses the JSR-175 default mechanism wherever possible, this mechanism is only suitable for defining defaults that are constant values. In contrast, many actual default values are not constants but are instead computed from the Java source or other annotations. For example, the default value for the @WebService.name annotation is the simple name of the Java class or interface. This value cannot be represented directly as a JSR-175 default. In scenarios where JSR-175 defaults are not sufficient to describe the required default, a "marker" constant is used instead. When the Jakarta Web Services Metadata processor encounters this marker constant, the processor treats the member-value as though it had the computed default described in section 4. For example,

when the Jakarta Web Services Metadata processor encounters a @WebService.name annotation with a value of "" (the empty string), it behaves as though the name of the Web Service were the name of the Java class.

2.6. Web Services Metadata

Jakarta Web Services Metadata Specification describes declaratively how the logic of a service implementation bean is exposed over networking protocols as a Web Service. The @WebService tag marks a Java class as implementing a Web Service. @WebMethod tags identify the individual methods of the Java class that are exposed externally as Web Service operations, as illustrated in the following example. The example uses JSR-175 syntax and the *annotation type* declarations defined in the jakarta.jws and jakarta.jws.soap packages.

```java
import jakarta.jws.WebService;
import jakarta.jws.WebMethod;

@WebService
public class HelloWorldService {
    @WebMethod
    public String helloWorld() {
      return "Hello World!";
    }
}
```

Most of these metadata tags have reasonable defaults, which are explicitly called out in section 4. Most of these metadata tags have reasonable defaults, which are explicitly called out in this document. The JWS author can avoid providing tags for the most common Web Service definitions.

Sections 2.6.1 through 2.6.3 describe the types of annotations provided by Jakarta Web Services Metadata Specification.

2.6.1. WSDL Mapping Annotations

WSDL mapping annotations control the mapping from Java source onto WSDL constructs. As described in section *2.2 Development Models*, this specification supports both a "start with Java" and a "start with WSDL" development model. In "start with Java," the WSDL mapping annotations control the shape of the WSDL generated from the Java source. In "start with WSDL," the WSDL mapping annotations associate the Java source with pre-existing WSDL constructs.

2.6.2. Binding Annotations

Binding annotations specify the network protocols and message formats that are supported by the Web Service. For example, the presence of a @SOAPBinding annotation tells the processor to make the service available over the SOAP 1.1 message. Fields on this annotation allow the developer to

customize the way the mapping of the implementation object onto SOAP messages.

Jakarta Web Services Metadata Specification defines a single set of annotations that map the implementation object to the SOAP protocol binding. Jakarta Web Services Metadata implementations MAY support additional binding annotations for other protocols. Non-normative examples of such binding annotations can be found in Appendix C.

2.6.3. Handler Annotations

Handler annotations allow the developer to extend a Web Service with additional functionality that runs before and after the business methods of the Web Service.

Chapter 3. Server Programming Model

This section describes the server programming model for Jakarta Web Services Metadata. The Jakarta Web Services Metadata server programming model is a simplification of the existing Jakarta Web Services server programming models, as defined in Jakarta XML Web Services and Jakarta Enterprise Web Services. Jakarta Web Services Metadata Specification simplifies these models by allowing the developer to focus on business logic and using annotations to generate related artifacts.

3.1. Service Implementation Bean

A developer who implements Web Services with Jakarta Web Services Metadata is responsible for implementing the service implementation bean containing the Web Service's business logic. A Jakarta Web Services Metadata service implementation bean MUST meet the following requirements:

- The implementation bean MUST be an outer public class, MUST NOT be final, and MUST NOT be abstract.

- The implementation bean MUST have a default public constructor.

- The implementation MUST NOT define a finalize() method.

- The implementation bean MUST include a @WebService class-level annotation, indicating that it implements a Web Service. More information on the @WebService annotation may be found in section *4.1 Annotation: jakarta.jws.WebService.*.

- The implementation bean MAY reference a service endpoint interface by using the @WebService.endpointInterface annotation. If the implementation bean references a service endpoint interface, it MUST implement all the methods on the service endpoint interface. If the implementation bean references a service endpoint interface, that service endpoint interface is used to determine the abstract WSDL contract (portType and bindings). In this case, the service implementation bean MUST NOT include any Jakarta Web Services Metadata annotations other than @WebService and @HandlerChain. In addition, the @WebService annotation MUST NOT include the name annotation element. More information on the @WebService.endpointInterface annotation element may be found in section *4.1 Annotation: jakarta.jws.WebService*.

- If the implementation bean does not reference a service endpoint interface by using the @WebService.endpointInterface annotation, the bean class implicitly defines a service endpoint interface (SEI). The SEI MUST meet the requirements specified in Jakarta XML Web Services Specification [5], section 3.3

3.2. Service Endpoint Interface

A Jakarta Web Services Metadata service implementation bean MAY reference a service endpoint interface, thus separating the contract definition from the implementation. A Jakarta Web Services Metadata service endpoint interface MUST meet the requirements specified in Jakarta XML Web Services Specification [5], section 3.4, with the following exceptions:

- The service endpoint interface MUST be an outer public interface.

- The service endpoint interface MUST include a @WebService annotation, indicating that it is defining the contract for a Web Service.

- The service endpoint interface MAY extend java.rmi.Remote either directly or indirectly, but is not REQUIRED to do so.

- All methods on the service endpoint interface, including methods inherited from super-interfaces, are mapped to WSDL operations regardless of whether they include a @WebMethod annotation. A method MAY include a @WebMethod annotation to customize the mapping to WSDL, but is not REQUIRED to do so.

- The service endpoint interface MAY include other Jakarta Web Services Metadata annotations to control the mapping from Java to WSDL.

- The service endpoint interface MUST NOT include the Jakarta Web Services Metadata annotation elements portName, serviceName and endpointInterface of the annotation @WebService.

3.3. Web Method

A method will be exposed as a Web Service operation, making it part of the Web Service's public contract according to rules specified in section *3.1 Service Implementation Bean* or in section *3.2 Service Endpoint Interface* if the service implementation bean implements a service endpoint interface. An exposed method MUST meet the following requirements.

- The method MUST be public.

- The method's parameters, return value, and exceptions MUST follow the rules defined in Jakarta XML Web Services Specification [5], section 3.6).

- The method MAY throw java.rmi.RemoteException, but is not REQUIRED to do so.

Chapter 4. Web Services Metadata

This section contains the specifications of each individual Web Service metadata items. Both the *annotation type* declarations (using JSR-175 syntax) and usage examples are given for each metadata item.

4.1. Annotation: jakarta.jws.WebService

4.1.1. Description

Marks a Java class as implementing a Web Service, or a Java interface as defining a Web Service interface.

Member-Value	Meaning	Default
name	The name of the Web Service. Used as the name of the wsdl:portType when mapped to WSDL 1.1	Simple name of the Java class or interface

Member-Value	Meaning	Default
targetNamespace	If the @WebService.targetNamespace annotation is on a service endpoint interface, the targetNamespace is used for the namespace for the wsdl:portType (and associated XML elements). If the @WebService.targetNamespace annotation is on a service implementation bean that does NOT reference a service endpoint interface (through the endpointInterface annotation element), the targetNamespace is used for both the wsdl:portType and the wsdl:service (and associated XML elements). If the @WebService.targetNamespace annotation is on a service implementation bean that does reference a service endpoint interface (through the endpointInterface annotation element), the targetNamespace is used for only the wsdl:service (and associated XML elements).	Implementation-defined, as described in Jakarta XML Web Services Specification [5], section 3.2.
serviceName	The service name of the Web Service. Used as the name of the wsdl:service when mapped to WSDL 1.1. This member-value is not allowed on endpoint interfaces.	Simple name of the Java class + "Service"

Member-Value	Meaning	Default
portName	Used as the name of the wsdl:port when mapped to WSDL 1.1. This member-value is not allowed on endpoint interfaces.	@WebService.name +"Port"
wsdlLocation	The location of a pre-defined WSDL describing the service. The wsdlLocation is a URL (relative or absolute) that refers to a pre-existing WSDL file. The presence of a wsdlLocation value indicates that the service implementation bean is implementing a pre-defined WSDL contract. The Jakarta Web Services Metadata tool MUST provide feedback if the service implementation bean is inconsistent with the portType and bindings declared in this WSDL. Note that a single WSDL file might contain multiple portTypes and multiple bindings. The annotations on the service implementation bean determine the specific portType and bindings that correspond to the Web Service.	None

Member-Value	Meaning	Default
endpointInterface	The complete name of the service endpoint interface defining the service's abstract Web Service contract. This annotation allows the developer to separate the interface contract from the implementation. If this annotation is present, the service endpoint interface is used to determine the abstract WSDL contract (portType and bindings). The service endpoint interface MAY include Jakarta Web Services Metadata annotations to customize the mapping from Java to WSDL. The service implementation bean MAY implement the service endpoint interface, but is not REQUIRED to do so. This member-value is not allowed on endpoint interfaces.	None. The Web Service contract is generated from annotations on the service implementation bean. If a service endpoint interface is required by the target environment, it will be generated into an implementation-defined package with an implementation-defined name.

4.1.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({TYPE})
public @interface WebService {
  String name() default "";
  String targetNamespace() default "";
  String serviceName() default "";
  String portName() default "";
  String wsdlLocation() default "";
  String endpointInterface() default "";
};
```

4.1.3. Example

Java source:

```
/**
* Annotated Implementation Object
*/
@WebService(
  name = "EchoService",
  targetNamespace = "http://www.openuri.org/2004/04/HelloWorld"
)
public class EchoServiceImpl {
    @WebMethod
    public String echo(String input) {
        return input;
    }
}
```

4.2. Annotation: jakarta.jws.WebMethod

4.2.1. Description

Customizes a method that is exposed as a Web Service operation. The WebMethod annotation includes the following member-value pairs:

Member-Value	Meaning	Default
operationName	Name of the wsdl:operation matching this method.	Name of the Java method
action	The action for this operation. For SOAP bindings, this determines the value of the soap action.	""
exclude	Marks a method to NOT be exposed as a web method. Used to stop an inherited method from being exposed as part of this web service. If this element is specified, other elements MUST NOT be specified for the @WebMethod. This member-value is not allowed on endpoint interfaces.	False

4.2.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface WebMethod {
  String operationName() default "";
  String action() default "" ;
  boolean exclude() default false;
};
```

4.2.3. Example

Java source:

```
@WebService
public class MyWebService {
    @WebMethod(operationName = "echoString", action="urn:EchoString")
    public String echo(String input) {
        return input;
    }
}
```

Resulting WSDL:

```
<definitions>
    <portType name="MyWebService">
        <operation name="echoString"/>
            <input message="echoString"/>
            <output message="echoStringResponse"/>
        </operation>
    </portType>

    <binding name="PingServiceHttpSoap" type="MyWebService">
        <operation name="echoString">
            <soap:operation soapAction="urn:EchoString"/>
        </operation>
    </binding>
</definitions>
```

4.3. Annotation: jakarta.jws.Oneway

4.3.1. Description

Indicates that the given web method has only an input message and no output. Typically, a oneway method returns the thread of control to the calling application prior to executing the actual business method. A Jakarta Web Services Metadata processor is REQUIRED to report an error if an operation marked @Oneway has a return value, declares any checked exceptions or has any INOUT or OUT parameters.

4.3.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface Oneway {
};
```

4.3.3. Example

Java source:

```
@WebService
public class PingService {

    @WebMethod
    @Oneway
    public void ping() {
    }
};
```

Resulting WSDL:

```
<definitions>
    <message name="ping"/>

    <portType name="PingService">
        <operation name="ping">
            <input message="ping"/>
        </operation>
    </portType>
</definitions>
```

4.4. Annotation: jakarta.jws.WebParam

4.4.1. Description

Customizes the mapping of an individual parameter to a Web Service message part and XML element.

Member-Value	Meaning	Default
name	Name of the parameter. If the operation is rpc style and @WebParam.partName has not been specified, this is name of the wsdl:part representing the parameter. If the operation is document style or the parameter maps to a header, this is the local name of the XML element representing the parameter. A name MUST be specified if the operation is document style, the parameter style is BARE, and the mode is OUT or INOUT.	@WebMethod.operation Name, if the operation is document style and the parameter style is BARE, and the parameter does not map to a header, and the mode is IN or INOUT. @WebMethod operation Name+"Response", if the operation is document style and the parameter style is BARE, and the parameter does not map to a header, and the mode is OUT. Otherwise, the default is arg$N$, where N represents the index of the parameter in the method signature (starting at arg0).
partName	The name of the wsdl:part representing this parameter. This is only used if the operation is rpc style or if the operation is document style and the parameter style is BARE.	@WebParam.name
targetNamespace	The XML namespace for the parameter. Only used if the operation is document style or the paramater maps to a header. If the target namespace is set to "", this represents the empty namespace.	The empty namespace, if the operation is document style, the parameter style is WRAPPED, and the parameter does not map to a header. Otherwise, the default is the targetNamespace for the Web Service.

Member-Value	Meaning	Default
mode	The direction in which the parameter is flowing. One of IN, OUT, or INOUT. The OUT and INOUT modes may only be specified for parameter types that conform to the definition of Holder types (Jakarta XML Web Services Specification [5], section 2.3.3). Parameters that are Holder Types MUST be OUT or INOUT.	IN if not a Holder type. INOUT if a Holder type.
header	If true, the parameter is pulled from a message header rather then the message body.	False

4.4.2. Annotation Type Definition

```java
@Retention(value=RetentionPolicy.RUNTIME)
@Target({PARAMETER})
public @interface WebParam {

    public enum Mode {
        IN,
        OUT,
        INOUT
    };

    String name() default "";
    String partName() default "";
    String targetNamespace() default "";
    Mode mode() default Mode.IN;
    boolean header() default false;
};
```

4.4.3. Example

Java Source:

```java
@WebService(targetNamespace="http://www.openuri.org/jwsm/WebParamExample")
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class PingService {

    @WebMethod(operationName = "PingOneWay")
    @Oneway
    public void ping(PingDocument ping) {
    }

    @WebMethod(operationName = "PingTwoWay")
    public void ping(
       @WebParam(mode=WebParam.Mode.INOUT)
          PingDocumentHolder ping) {
    }

    @WebMethod(operationName = "SecurePing")
    @Oneway
    public void ping(
       PingDocument ping,
       @WebParam(header=true)
          SecurityHeader secHeader) {
    }
};
```

Resulting WSDL:

```xml
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.openuri.org/jwsm/WebParamExample"
  xmlns:wsdl="http://www.openuri.org/jwsm/WebParamExample"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://www.openuri.org/jwsm/WebParamExample">

  <types>
     <s:schema elementFormDefault="qualified"
targetNamespace="http://www.openuri.org/jwsm/WebParamExample">
        <s:complexType name="PingDocument">
           . . .
        </s:complexType>
        <s:complexType name="SecurityHeader">
           . . .
        </s:complexType>
        <s:element name="SecurityHeader" type="SecurityHeader"/>
     </s:schema>
  </types>
```

```xml
<message name="PingOneWay">
    <part name="arg0" type="tns:PingDocument"/>
</message>

<message name="PingTwoWay">
    <part name="arg0" type="tns:PingDocument"/>
</message>

<message name="PingTwoWayResponse">
    <part name="arg0" type="tns:PingDocument"/>
</message>

<message name="SecurePing">
    <part name="arg0" type="tns:PingDocument"/>
    <part name="arg1" element="tns:SecurityHeader"/>
</message>

<portType name="PingService">
    <operation name="PingOneWay">
        <input message="tns:PingOneWay"/>
    </operation>

    <operation name="PingTwoWay">
        <input message="tns:PingTwoWay"/>
        <output message="tns:PingTwoWayResponse"/>
    </operation>

    <operation name="SecurePing">
        <input message="tns:SecurePing"/>
    </operation>
</portType>

<binding name="PingServiceHttpSoap" type="tns:PingService">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    <operation name="PingOneWay">
        <soap:operation soapAction="http://openuri.org/PingOneWay"/>
        <input>
            <soap:body parts="arg0" use="literal"/>
        </input>
    </operation>

    <operation name="PingTwoWay">
        <soap:operation soapAction="http://openuri.org/PingTwoWay"/>
        <input>
            <soap:body parts="arg0" use="literal"/>
        </input>
        <output>
```

```
            <soap:body parts="arg0" use="literal"/>
        </output>
    </operation>

    <operation name="SecurePing">
        <soap:operation soapAction="http://openuri.org/SecurePing"/>
        <input>
            <soap:body parts="arg0" use="literal"/>
            <soap:header message="SecurePing" part="arg1" use="literal"/>
        </input>
    </operation>
  </binding>
</definitions>
```

4.5. Annotation: jakarta.jws.WebResult

4.5.1. Description

Customizes the mapping of the return value to a WSDL part and XML element.

Member-Value	Meaning	Default
name	Name of return value. If the operation is rpc style and @WebResult.partName has not been specified, this is the name of the wsdl:part representing the return value. If the operation is document style or the return value maps to a header, this is the local name of the XML element representing the return value.	@WebParam.operation Name+"Response," if the operation is document style and the parameter style is BARE. Otherwise, the default is "return."
partName	The name of the wsdl:part representing this return value. This is only used if the operation is rpc style, or if the operation is document style and the parameter style is BARE.	@WebResult.name

Member-Value	Meaning	Default
targetNamespace	The XML namespace for the return value. Only used if the operation is document style or the return value maps to a header. If the target namespace is set to " ", this represents the empty namespace.	The empty namespace, if the operation is document style, the parameter style is WRAPPED, and the return value does not map to a header, Otherwise, the default is the targetNamespace for the Web Service.
header	If true, the parameter is in the message header rather then the message body.	False

4.5.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({METHOD})
public @interface WebResult {
  String name() default "";
  String partName() default "";
  String targetNamespace() default "";
  boolean header() default false;
};
```

4.5.3. Example

Java Source:

```
@WebService
public class CustomerService {

   @WebMethod
   @WebResult(name="CustomerRecord")
   public CustomerRecord locateCustomer(
      @WebParam(name="FirstName") String firstName,
      @WebParam(name="LastName") String lastName,
      @WebParam(name="Address") USAddress addr) {
   }
};
```

Resulting WSDL:

```
<definitions>
    <types>
        <complexType name="CustomerRecord">
            ...
        </complexType>

        <complexType name="USAddress">
            ...
        </complexType>

        <element name="locateCustomer">
          <complexType>
            <sequence>
              <element name="FirstName" type="xs:string"/>
              <element name="LastName" type="xs:string"/>
              <element name="Address" type="USAddress"/>
            </sequence>
          </complexType>
        </element>

        <element name="locateCustomerResponse">
          <complexType>
            <sequence>
              <element name="CustomerRecord" type="CustomerRecord"/>
            </sequence>
          </complexType>
        </element>
    </types>

    <message name="locateCustomer">
        <part name="parameters" element="tns:locateCustomer"/>
    </message>

    <message name="locateCustomerResponse">
        <part name="parameters" element="tns:locateCustomerResponse"/>
    </message>

    <portType name="CustomerService">
        <operation name="locateCustomer">
            <input message="tns:locateCustomer"/>
            <output message="tns:locateCustomerResponse"/>
        </operation>
    </portType>
</definitions>
```

4.6. Annotation: jakarta.jws.HandlerChain

4.6.1. Description

The @HandlerChain annotation associates the Web Service with an externally defined handler chain (Jakarta XML Web Services Specification [5], Section 9).

It is an error to combine this annotation with the @SOAPMessageHandlers annotation.

The @HandlerChain annotation MAY be present on the endpoint interface and service implementation bean. The service implementation bean's @HandlerChain is used if @HandlerChain is present on both.

The @HandlerChain annotation MAY be specified on the type only. The annotation target includes METHOD and FIELD for use by Jakarta XML Web Services Specification [5]. A Jakarta Web Services Metadata Processor is REQUIRED to report an error if the @HanderChain annotation is used on a method.

The @HandlerChain annotation contains the following member-values:

Member-Value	Meaning	Default
File	Location of the handler chain file. The location supports 2 formats. 1. An absolute java.net.URL in externalForm. (ex: http://myhandlers.foo.com/handlerfile1.xml) 2. A relative path from the source file or class file. (ex: bar/handlerfile1.xml)	None

Member-Value	Meaning	Default
name	**Deprecated** as of Jakarta Web Services Metadata 2.0 with no replacement. The name was originally used to associate a Jakarta XML RPC handler in a handler chain with the web service it is declared in. Jakarta XML Web Services handlers are associated to Web Services through elements in the handler chain itself. In this version, the name is ALWAYS ignored. This member-value will be permanently removed in a future version of Jakarta Web Services Metadata Specification.	””

4.6.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({TYPE, METHOD, FIELD})
public @interface HandlerChain {
  String file();
  String name() default "";
};
```

4.6.3. Examples

Example 1

Java Source:

Located in /home/mywork/src/com/jwsm/examples/

```java
package com.jwsm.examples;

@WebService
@HandlerChain(file="config/ProjectHandlers.xml")
public class MyWebService {
};
```

Handler Chain Configuration File

Located in /home/mywork/src/com/jwsm/examples/config/

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<bindings
  wsdlLocation="http://localhost:8080/fromwsdl_handler/test?wsdl"
  xmlns="https://jakarta.ee/xml/ns/jaxws">

  <bindings node="ns1:definitions" xmlns:ns1="http://schemas.xmlsoap.org/wsdl/">
    <package name="fromwsdl.handler.client"/>
  </bindings>

  <bindings node="ns1:definitions/ns1:types/xs:schema[@targetNamespace='urn:test:types']"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:ns1="http://schemas.xmlsoap.org/wsdl/">
    <ns2:schemaBindings xmlns:ns2="https://jakarta.ee/xml/ns/jaxb">
       <ns2:package name="fromwsdl.handler.client"/>
    </ns2:schemaBindings>
  </bindings>

  <bindings>
    <handler-chains xmlns="https://jakarta.ee/xml/ns/jakartaee">
      <handler-chain>
        <handler>
          <handler-class>fromwsdl.handler.common.BaseLogicalHandler</handler-class>
          <init-param>
            <param-name>handlerName</param-name>
            <param-value>client0</param-value>
          </init-param>
        </handler>
      </handler-chain>
      <handler-chain>
        <port-name-pattern xmlns:ns2="urn:test">ns2:Report</port-name-pattern>
        <handler>
          <handler-class>fromwsdl.handler.common.BaseLogicalHandler</handler-class>
          <init-param>
            <param-name>handlerName</param-name>
            <param-value>client2</param-value>
```

```
        </init-param>
      </handler>
    </handler-chain>
    <handler-chain>
      <port-name-pattern xmlns:ns2="urn:test">ns2:ReportServicePort</port-name-pattern>
      <handler>
        <handler-class>fromwsdl.handler.common.BaseSOAPHandler</handler-class>
        <init-param>
          <param-name>handlerName</param-name>
          <param-value>client6</param-value>
        </init-param>
      </handler>
    </handler-chain>
    <handler-chain>
      <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
      <handler>
        <handler-class>fromwsdl.handler.common.BaseSOAPHandler</handler-class>
        <init-param>
          <param-name>handlerName</param-name>
          <param-value>client7</param-value>
        </init-param>
        <soap-role>http://sun.com/client/role1</soap-role>
        <soap-role>http://sun.com/client/role2</soap-role>
      </handler>
    </handler-chain>
    <handler-chain>
      <protocol-bindings>##SOAP11_HTTP</protocol-bindings>
      <handler>
        <handler-class>fromwsdl.handler.common.BaseLogicalHandler</handler-class>
        <init-param>
          <param-name>handlerName</param-name>
          <param-value>client3</param-value>
        </init-param>
      </handler>
    </handler-chain>
  </handler-chains>
  </bindings>
</bindings
```

4.7. Annotation: jakarta.jws.soap.SOAPBinding

4.7.1. Description

Specifies the mapping of the Web Service onto the SOAP message protocol. Section *6 SOAP Binding* describes the effects of this annotation on generated Web Services. The SOAPBinding annotation has a target of TYPE and METHOD. The annotation may be placed on a method if and only if the

SOAPBinding.style is DOCUMENT. Implementations MUST report an error if the SOAPBinding annotation is placed on a method with a SOAPBinding.style of RPC. Methods that do not have a SOAPBinding annotation accept the SOAPBinding behavior defined on the type.

The @SOAPBinding annotation includes the following member-value pairs.

Member-Value	Meaning	Default
style	Defines the encoding style for messages send to and from the Web Service. One of DOCUMENT or RPC.	DOCUMENT
use	Defines the formatting style for messages sent to and from the Web Service. One of LITERAL or ENCODED.	LITERAL
parameterStyle	Determines whether method parameters represent the entire message body, or whether the parameters are elements wrapped inside a top-level element named after the operation.	WRAPPED

4.7.2. Annotation Type Definition

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target({TYPE, METHOD})
public @interface SOAPBinding {
    public enum Style {
        DOCUMENT,
        RPC
    };

    public enum Use {
        LITERAL,
        ENCODED
    };

    public enum ParameterStyle {
        BARE,
        WRAPPED
    }

    Style style() default Style.DOCUMENT;
    Use use() default Use.LITERAL;
    ParameterStyle parameterStyle() default ParameterStyle.WRAPPED;
}
```

4.7.3. Examples

Example 1 – RPC/LITERAL

Java source:

```
@WebService(targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample1")
@SOAPBinding(
    style = SOAPBinding.Style.RPC,
    use = SOAPBinding.Use.LITERAL)
public class ExampleService {
    @WebMethod
    public String concat(String first, String second, String third) {
        return first + second + third;
    }
}
```

Resulting WSDL:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.openuri.org/jwsm/SoapBindingExample1"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample1">

  <message name="concat">
     <part name="first" type="xs:string"/>
     <part name="second" type="xs:string"/>
     <part name="third" type="xs:string"/>
  </message>

  <message name="concatResponse">
     <part name="return" type="xs:string"/>
  </message>

  <portType name="ExampleService">
     <operation name="concat">
       <input message="tns:concat"/>
       <output message="tns:concatResponse"/>
     </operation>
  </portType>

  <binding name="ExampleServiceHttpSoap" type="ExampleService">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="concat">
      <soap:operation soapAction="http://www.openuri.org/jwsm/SoapBindingExample1/concat
"/>
      <input>
        <soap:body parts="first second third" use="literal"/>
      </input>
      <output>
        <soap:body parts="return" use="literal"/>
      </output>
    </operation>
   </binding>
</definitions>
```

Example 2 – DOCUMENT/LITERAL/BARE

Java source:

```java
@WebService(targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample2")
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class DocBareService {

    @WebMethod( operationName="SubmitPO" )
    public SubmitPOResponse submitPO(SubmitPORequest submitPORequest) {
    }
}
```

Resulting WSDL:

```xml
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.openuri.org/jwsm/SoapBindingExample2"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample2">

  <types>
      <s:schema elementFormDefault="qualified" targetNamespace=
"http://www.openuri.org/jwsm/SoapBindingExample2">
          <s:element name="SubmitPORequest">
              . . .
          </s:element>
          <s:element name="SubmitPOResponse">
              . . .
          </s:element>
      </s:schema>
  </types>

  <message name="SubmitPO">
      <part name="parameters" element="tns:SubmitPORequest"/>
  </message>

  <message name="SubmitPOResponse">
      <part name="parameters" element="tns:SubmitPOResponse"/>
  </message>

  <portType name="DocBareService">
      <operation name="SubmitPO">
          <input message="tns:SubmitPO"/>
          <output message="tns:SubmitPOResponse"/>
      </operation>
  </portType>

  <binding name="DocBareServiceHttpSoap" type="ExampleService">
```

```xml
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="SubmitPO">
            <soap:operation soapAction=
"http://www.openuri.org/jwsm/SoapBindingExample2/SubmitPO />
            <input>
                <soap:body parts="parameters" use="literal"/>
            </input>
            <output>
                <soap:body parts="parameters" use="literal"/>
            </output>
        </operation>
    </binding>
</definitions>
```

Example 3 – DOCUMENT/LITERAL/WRAPPED

Java source:

```java
@WebService(targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample3")
@SOAPBinding(
  style = SOAPBinding.Style.DOCUMENT,
  use = SOAPBinding.Use.LITERAL,
  parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)
public class DocWrappedService {

    @WebMethod(operationName = "SubmitPO")
    @WebResult(name="PurchaseOrderAck")
    public PurchaseOrderAck submitPO(
        @WebParam(name="PurchaseOrder") PurchaseOrder purchaseOrder) {
    }
}
```

Resulting WSDL:

```xml
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://www.openuri.org/jwsm/SoapBindingExample3"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://www.openuri.org/jwsm/SoapBindingExample3">

  <types>
    <s:schema elementFormDefault="qualified" targetNamespace=
"http://www.openuri.org/jwsm/SoapBindingExample3">
        <s:element name="SubmitPO">
          <complexType>
```

```
            <sequence>
              <element name="PurchaseOrder" type="tns:PurchaseOrder"/>
                  . . .
        </s:element>

        <s:element name="SubmitPOResponse">
            . . .
        </s:element>

    </s:schema>
  </types>

  <message name="SubmitPO">
      <part name="parameters" element="tns:SubmitPO"/>
  </message>

  <message name="SubmitPOResponse">
      <part name="parameters" type="tns:SubmitPOResponse"/>
  </message>

  <portType name="DocWrappedService">
      <operation name="SubmitPO">
          <input message="tns:SubmitPO"/>
          <output message="tns:SubmitPOResponse"/>
      </operation>
  </portType>

  <binding name="ExampleServiceHttpSoap" type="ExampleService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="SubmitPO">
      <soap:operation soapAction=
"http://www.openuri.org/jwsm/SoapBindingExample3/SubmitPO" />
      <input>
        <soap:body parts="parameters" use="literal"/>
      </input>
      <output>
        <soap:body parts="parameters" use="literal"/>
      </output>
    </operation>
  </binding>
</definitions>
```

4.8. Annotation: jakarta.jws.soap.SOAPMessageHandlers

Deprecated as of Jakarta Web Services Metadata 2.0 with no replacement.

This annotation was originally used to create a Jakarta XML RPC handler chain. In this version, the annotation is ALWAYS ignored.

This annotation will be permanently removed in a future version of Jakarta Web Services Metadata Specification.

Chapter 5. Java Mapping To XML/WSDL

A key goal of Jakarta Web Services Metadata is to influence the shape of WSDL generated from a JWS. This section defines the mapping from Java to XML/WSDL. By default, Jakarta Web Services Metadata follows the Java to XML/WSDL mapping defined in Jakarta XML Web Services Specification [5] section 3, except as noted in this section. Implementations MAY extend or supplement this mapping, for example, by adding more complete schema support or supporting alternate binding frameworks such as Jakarta XML Binding or SDO (JSR-235). Annotations for such extensions are out-of-scope for this specification.

5.1. Service Endpoint Interface

Jakarta XML Web Services defines a service endpoint interface as the Java representation of an abstract WSDL contract. A service endpoint interface MAY include the following Jakarta Web Services Metadata annotations to customize its mapping to WSDL:

- @WebService.name, @WebService.targetNamespace, and @WebService.wsdlLocation

- @WebMethod (all annotation elements)

- @Oneway

- @WebParam (all annotation elements)

- @WebResult (all annotation elements)

- @SOAPBinding (all annotation elements)

A service endpoint interface maps to a wsdl:portType element within the wsdl:definitions for the containing package. The local name and namespace of the wsdl:portType map to the values of the service endpoint interface's @WebService.name and @WebService.targetNamespace annotation elements, respectively.

5.2. Web Service Class Mapping

A service implementation bean maps to its own WSDL document, wsdl:portType, and wsdl:service. If the service implementation bean references a service endpoint interface through the @WebService.endpointInterface annotation, the wsdl:portType and wsdl:binding sections are mapped according to that service endpoint interface. Otherwise, the following rules apply:

- The wsdl:definitions targetNamespace maps to the value of the @WebService.targetNamespace member-value.

- The local name of the wsdl:portType maps to the value of the @WebService.name member-value.

- The local name of the wsdl:service maps to the value of the @WebService.serviceName member-value.

- The wsdl:service MUST contain a distinct wsdl:port for every transport endpoint supported by the

service.

- Each wsdl:port MUST be of the same wsdl:portType, but MAY have different bindings.

- The local name of the wsdl:port maps to the value of the @WebService.portName member-value.

- The name wsdl:binding sections is not significant and are left implementation-defined.

5.3. Web Method Mapping

Each exposed web method in a Jakarta Web Services Metadata annotated class or interface is mapped to a wsdl:operation on the class/interface WSDL portType. The wsdl:operation local name maps to the value of the @WebMethod.operationName member-value, if @WebMethod.operationName is present. If @WebMethod.operationName is not present, the wsdl:operation local name is mapped from the name of the Java method according to the rules defined in Jakarta XML Web Services Specification [5], section 3.5.

The mapped wsdl:operation contains both wsdl:input and wsdl:output elements, unless the method is annotated as @Oneway. @Oneway methods have only a wsdl:input element.

Java types used as method parameters, return values, and exceptions are mapped according to the rules defined in Jakarta XML Web Services [5], section 3.6.

Chapter 6. SOAP Binding

This section defines a standard mapping from a service endpoint interface or service implementation bean to the SOAP 1.1 binding. Implementers MAY also support other bindings, but these bindings are non-standard. If Jakarta Web Services Metadata implementation supports bindings other than SOAP 1.1, it MUST include a mechanism to selectively enable or disable these bindings.

By default Jakarta Web Services Metadata Specification follows the SOAP binding defined in Jakarta XML Web Services Specification [5], section 10.

6.1. Operation Modes

Jakarta Web Services Metadata implementations are REQUIRED to support the following WS-I compliant operation modes:

- Operations with the rpc style and literal use (rpc/literal)
- Operations with the document style and literal use (document/literal).

Implementations MAY optionally support operation modes with the encoded use (document or rpc style). The developer MAY indicate which operation mode is in effect by specifying the appropriate @SOAPBinding.style and @SOAPBinding.use annotations at the class or interface level.

6.1.1. RPC Operation Style

In the RPC operation style, the parameters and return values map to separate parts on the WSDL input and output messages. The @WebParam.mode annotation determines the messages in which a particular parameter appears. IN parameters appear as parts in the input message, OUT parameters appear as parts in the output message, and INOUT parameters appear as parts in both messages. The order of parameters in the method signature determines the order of the parts in the input and output message. The return value is the first part in the output message.

In the rpc/literal operation mode, each message part refers to a concrete schema type. The schema type is derived from the Java type for the parameter, as described in section *5 Java Mapping To XML/WSDL*.

6.1.2. Document Operation Style

In the document operation style, the input and output WSDL messages have a single part referencing a schema element that defines the entire body. Jakarta Web Services Metadata implementations MUST support both the "wrapped" and "bare" styles of document / literal operation. The developer may specify which of these styles is in effect for a particular operation by using the @SOAPBinding.parameterStyle annotation.

6.1.3. Document "Wrapped" Style

In the "wrapped" operation style, the input and output messages contain a single part which refers (through the *element* attribute) to a global element declaration (the *wrapper*) of complexType defined using the xsd:sequence compositor. The global element declaration for the input message has a local name equal to @WebMethod.operationName. The global element declaration for the output message (if it exists) has a local name equal to @WebMethod.operationName + "Response". Both global element declarations appear in the @WebService.targetNamespace.

Non-header method parameters and return values map to child elements of the global element declarations defined for the method. The order of parameters in the parameter list determines the order in which the equivalent child elements appear in the operation's global element declarations.

The @WebParam.name and @WebParam.targetNamespace annotation elements determine the QName of a parameter's child element, while the @WebResult.name and @WebResult.targetNamespace annotations determines the QName of the return value's child element. The schema type for each child element is derived from the type of the Java parameter or return value, as described in section *5 Java Mapping To XML/WSDL*.

6.1.4. Document "Bare" Style

In the "bare" operation style, the input and output messages contain a single part which refers (through the *element* attribute) to an element that is mapped from the method parameter and return value. The QName of the input body element is determined by the values of the @WebParam.name and @WebParam.targetNamespace annotations on the method parameter, and the QName of the output body element is determined by the values of the @WebResult.name and @WebResult.targetNamespace annotations. The schema types for the input and output body elements are derived from the types of the Java parameter or return values, as described in section *5 Java Mapping To XML/WSDL*.

Web Services that use the document "bare" style MUST adhere to the following restrictions:

- If the operation is marked @Oneway, it MUST have a void return value, a single non-header parameter marked as IN, and zero or more header parameters.

- If the operation is not marked @Oneway, it may have one of the following forms:

- A non-header parameter marked as IN, a non-header parameter marked as OUT, a void return value, and zero or more header parameters.

- A single non-header parameters marked as IN_OUT, a void return value, and zero or more header parameters.

- A single non-header parameter marked as IN, non-void return value and zero or more header parameters.

- The XML elements for the input and output messages MUST be unique across all operations on the Web Service. Consequently, either every document "bare" operation on the Web Service MUST take and return Java types that map to distinct elements, or the developer MUST use the @WebParam and @WebResult annotations to explicitly specify the QNames of the input and output XML

elements for each operation.

6.2. Headers

Parameters annotated with the @WebParam.header annotation element map to SOAP headers instead of elements in the SOAP body. Header parameters appear as parts in the operation's input message, output message, or both depending on the value of the @WebParam.mode annotation element. Header parameters are included as soap:header elements in the appropriate wsdl:input and wsdl:output sections of the binding operation. Headers are always literal. The @WebParam.name and @WebParam.targetNamespace annotations determine the QName of the XML element representing the header.

Results annotated with the @WebResult.header annotation element map to SOAP headers instead of elements in the SOAP body. Header results appear as parts in the operation's output message. Header results are included as soap:header elements in the appropriate wsdl:output sections of the binding operation. Headers are always literal. The @WebResult.name and @WebResult.targetNamespace annotations determine the QName of the XML element representing the header. This QName MUST be unique within all headers of the method.

Chapter 7. Using Jakarta Web Services Metadata Annotations to Affect the Shape of the WSDL

7.1. RPC Literal Style

Below is a complete example of a java source file with annotations followed by the resulting WSDL:

Java source:

```java
import jakarta.jws.*;
import jakarta.jws.soap.*;

@WebService(
  name="ExampleWebService",
  targetNamespace="http://openuri.org/11/2003/ExampleWebService")
@SOAPBinding(style=SOAPBinding.Style.RPC, use=SOAPBinding.Use.LITERAL)
public class ExampleWebServiceImpl {

  @WebMethod(action="urn:login")
  @WebResult(name="Token")
  public LoginToken login(
      @WebParam(name="UserName") String username,
      @WebParam(name="Password") String password) {
    // ...
  }

  @WebMethod (action="urn:createCustomer")
  @WebResult(name="CustomerId")
  public String createCustomer(
      @WebParam(name="Customer") Customer customer,
      @WebParam(name="Token", header=true) LoginToken token) {
    // ...
  }

  @WebMethod(action="urn:notifyTransfer")
  @Oneway
  public void notifyTransfer(
      @WebParam(name="CustomerId") String customerId,
      @WebParam(name="TransferData") TransferDocument transferData,
      @WebParam(name="Token", header=true) LoginToken token) {
  }
};
```

Resulting WSDL:

```
<definitions
  name="ExampleWebServiceImplServiceDefinitions"
  targetNamespace="http://openuri.org/11/2003/ExampleWebService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://openuri.org/11/2003/ExampleWebService"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

  <types>
    <xs:schema elementFormDefault="qualified"
        targetNamespace="http://openuri.org/11/2003/ExampleWebService">

      <xs:complexType name="LoginToken">
          ...
      </xs:complexType>

      <xs:complexType name="Customer">
          ...
      </xs:complexType>

      <xs:complexType name="TransferDocument">
          ...
      </xs:complexType>

      <xs:element name="Token" type="LoginToken"/>

    </xs:schema>
  </types>

  <message name="createCustomer">
    <part name="Customer" type="tns:Customer"/>
    <part element="tns:Token" name="token"/>
  </message>

  <message name="createCustomerResponse">
    <part name="CustomerId" type="xs:string"/>
  </message>

  <message name="notifyTransfer">
    <part name="CustomerId" type="xs:string"/>
    <part name="TransferData" type="tns:TransferDocument"/>
    <part name="token" element="tns:Token"/>
  </message>

  <message name="login">
```

```
    <part name="UserName" type="xs:string"/>
    <part name="Password" type="xs:string"/>
</message>

<message name="loginResponse">
    <part name="Token" type="tns:LoginToken"/>
</message>

<portType name="ExampleWebService">
    <operation name="createCustomer" parameterOrder="Customer token">
        <input message="tns:createCustomer"/>
        <output message="tns:createCustomerResponse"/>
    </operation>

    <operation name="notifyTransfer" parameterOrder="CustomerId TransferData token">
        <input message="tns:notifyTransfer"/>
    </operation>

    <operation name="login" parameterOrder="UserName Password">
        <input message="tns:login"/>
        <output message="tns:loginResponse"/>
    </operation>

</portType>

<binding name="ExampleWebServiceImplServiceSoapBinding"
        type="tns:ExampleWebService">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>

    <operation name="createCustomer">
        <soap:operation soapAction="urn:createCustomer" style="rpc"/>
        <input>
            <soap:body
                namespace="http://openuri.org/11/2003/ExampleWebService"
                parts="Customer"
                use="literal"/>
            <soap:header
                message="tns:createCustomer"
                part="token"
                use="literal"/>
        </input>
        <output>
            <soap:body
                namespace="http://openuri.org/11/2003/ExampleWebService"
                parts="CustomerId"
                use="literal"/>
        </output>
```

```
    </operation>

    <operation name="notifyTransfer">
      <soap:operation soapAction="urn:notifyTransfer" style="rpc"/>
      <input>
        <soap:body
            namespace="http://openuri.org/11/2003/ExampleWebService"
            parts="CustomerId TransferData"
            use="literal"/>
        <soap:header
            message="tns:notifyTransfer"
            part="token"
            use="literal"/>
      </input>
    </operation>

    <operation name="login">
      <soap:operation soapAction="urn:login" style="rpc"/>
      <input>
        <soap:body
            namespace="http://openuri.org/11/2003/ExampleWebService"
            parts="UserName Password"
            use="literal"/>
      </input>
      <output>
        <soap:body
            namespace="http://openuri.org/11/2003/ExampleWebService"
            parts="Token"
            use="literal"/>
      </output>
    </operation>
  </binding>

  <service name="ExampleWebServiceImplService">
    <port
        binding="s1:ExampleWebServiceImplServiceSoapBinding"
        name="ExampleWebServiceSoapPort">
      <soap:address
        location="http://localhost:7001/ExampleWebServiceImpl/ExampleWebServiceImpl"/>
    </port>
  </service>
</definitions>
```

7.2. Document Literal Style

Below is a complete example of a java source file with annotations followed by the resulting WSDL:

Java source:

```java
import jakarta.jws.*;
import jakarta.jws.soap.*;

@WebService(
  name="ExampleWebService",
  targetNamespace="http://openuri.org/11/2003/ExampleWebService")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
  use=SOAPBinding.Use.LITERAL)
public class ExampleWebServiceImpl {

  @WebMethod(action="urn:login")
  @WebResult(name="Token")
  public LoginToken login(
      @WebParam(name="UserName") String username,
      @WebParam(name="Password") String password) {
    // ...
  }

  @WebMethod (action="urn:createCustomer")
  @WebResult(name="CustomerId")
  public String createCustomer(
      @WebParam(name="Customer") Customer customer,
      @WebParam(name="Token", header=true) LoginToken token) {
    // ...
  }

  @WebMethod(action="urn:notifyTransfer")
  @Oneway
  public void notifyTransfer(
      @WebParam(name="CustomerId") String customerId,
      @WebParam(name="TransferData") TransferDocument transferData,
      @WebParam(name="Token", header=true) LoginToken token) {
  }

};
```

Resulting WSDL:

```xml
<?xml version='1.0' encoding='UTF-8'?>
<definitions
  name="ExampleWebServiceImplServiceDefinitions"
  targetNamespace="http://openuri.org/11/2003/ExampleWebService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://openuri.org/11/2003/ExampleWebService"
```

```
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

<types>
  <xs:schema attributeFormDefault="unqualified"
      targetNamespace="http://openuri.org/11/2003/ExampleWebService">

    <xs:complexType name="LoginToken">
        ...
    </xs:complexType>

    <xs:complexType name="Customer">
        ...
    </xs:complexType>

    <xs:complexType name="TransferDocument">
        ...
    </xs:complexType>

    <xs:element name="Token" type="tns:LoginToken"/>

    <xs:element name="createCustomer">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Customer" type="tns:Customer"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="createCustomerResponse">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="CustomerId" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="notifyTransfer">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="CustomerId" type="xs:string"/>
          <xs:element name="TransferData" type="tns:TransferDocument"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="login">
      <xs:complexType>
```

```xml
          <xs:sequence>
            <xs:element name="UserName" type="xs:string"/>
            <xs:element name="Password" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>

      <xs:element name="loginResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Token" type="tns:LoginToken"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
</types>

<message name="createCustomer">
  <part element="tns:createCustomer" name="parameters"/>
  <part element="tns:Token" name="token"/>
</message>

<message name="createCustomerResponse">
  <part element="tns:createCustomerResponse" name="parameters"/>
</message>

<message name="notifyTransfer">
  <part element="tns:notifyTransfer" name="parameters"/>
  <part element="tns:Token" name="token"/>
 </message>

<message name="login">
  <part element="tns:login" name="parameters"/>
</message>

<message name="loginResponse">
  <part element="tns:loginResponse" name="parameters"/>
</message>

<portType name="ExampleWebService">
  <operation name="createCustomer" parameterOrder="parameters token">
    <input message="tns:createCustomer"/>
    <output message="tns:createCustomerResponse"/>
  </operation>

  <operation name="notifyTransfer" parameterOrder="token">
    <input message="tns:notifyTransfer"/>
  </operation>
```

```xml
      <operation name="login" parameterOrder="parameters">
        <input message="tns:login"/>
        <output message="tns:loginResponse"/>
      </operation>
   </portType>

   <binding name="ExampleWebServiceImplServiceSoapBinding" type="tns:ExampleWebService">
     <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
     <operation name="createCustomer">
       <soap:operation soapAction="urn:createCustomer" style="document"/>
       <input>
         <soap:body parts="parameters" use="literal"/>
         <soap:header message="tns:createCustomer" part="token" use="literal"/>
       </input>
       <output>
         <soap:body parts="parameters" use="literal"/>
       </output>
     </operation>

     <operation name="notifyTransfer">
       <soap:operation soapAction="urn:notifyTransfer" style="document"/>
       <input>
         <soap:body parts="parameters" use="literal"/>
         <soap:header message="tns:notifyTransfer" part="token" use="literal"/>
       </input>
     </operation>

     <operation name="login">
       <soap:operation soapAction="urn:login" style="document"/>
       <input>
         <soap:body parts="parameters" use="literal"/>
       </input>
       <output>
         <soap:body parts="parameters" use="literal"/>
       </output>
     </operation>
   </binding>

   <service name="ExampleWebServiceImplService">
     <port binding="tns:ExampleWebServiceImplServiceSoapBinding" name=
"ExampleWebServiceSoapPort">
       <soap:address
           location="http://localhost:7001/ExampleWebServiceImpl/ExampleWebServiceImpl"/>
     </port>
</service>
</definitions>
```

Chapter 8. References

1. JSR-175 A Metadata Facility for the Java™ Programming Language https://jcp.org/en/jsr/detail?id=175

2. Jakarta Deployment 1.7 https://jakarta.ee/specifications/deployment/1.7/

3. XML Schema 1.0 https://www.w3.org/TR/xmlschema-1/

4. Jakarta EE 9 https://jakarta.ee/specifications/platform/9/

5. Jakarta XML Web Services 3.0 https://jakarta.ee/specifications/xml-web-services/3.0/

6. Jakarta Enterprise Web Services 2.0 https://jakarta.ee/specifications/enterprise-ws/2.0/

7. Jakarta Annotations 2.0 https://jakarta.ee/specifications/annotations/2.0/

8. Web Services Definition Language (WSDL) 1.1 https://www.w3.org/TR/wsdl/

9. Simple Object Access Protocol (SOAP) 1.1 https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

10. Apache AXIS "JWS" drop-in deployment of Web Services

11. BEA WebLogic Workshop "JWS" annotated Java Web Services

12. RFC 2119: Keywords for use in RFCs to Indicate Requirement Levels https://www.ietf.org/rfc/rfc2119.txt

Appendix A: Relationship to Other Standards

Jakarta Web Services Metadata relies on Java standards, Web Services standards, XML standards and Internet standards.

Java Language standards: J2SE 5.0 is needed for the JSR-175 defined Metadata Facility.

Java runtime and container standards: Jakarta Web Services Metadata does not define a container or runtime environment – implementers provide tools to map the Java classes to specific runtime environments. The functionality of the Jakarta containers is assumed. The features provided by Jakarta XML Web Services 2.3 are needed for the Web Services runtime as well as the mapping conventions; Java to XML/WSDL and WSDL/XML to Java. An optional mapping to Jakarta Enterprise Web Services deployment descriptors is provided in Jakarta Web Services Metadata.

Web Services standards: SOAP 1.1 and WSDL 1.1 are used to describe the Web Service and define the XML messages.

XML standards: The XML language and the XML Schema 1.0 are an integral part of Jakarta Web Services Metadata.

Internet standards: HTTP and HTTP/S provide basic protocols for Web Services.

Appendix B: Handler Chain Configuration File Schema

The XML Schema for the handler chain configuration file is described at https://jakarta.ee/xml/ns/ jakartaee/jakartaee_web_services_metadata_handler_3_0.xsd

Appendix C: Non-Normative Examples of Alternate Binding Annotations

This section defines non-normative examples of annotations for bindings to non-standard protocols and transports.

C.1 Annotation Name: HttpGetBinding

C.1.1 Description

Non-normative example of an alternate binding – in this case a raw HTTP binding as specified in WSDL 1.1 [7] section 4.

Member-Value	Meaning	Default
location	The location of the HTTP GET endpoint. When defined at the class level, defines as the base URI for all operations on the service. When defined at the method level, defines the URI for a particular operation relative to the base URI for the service.	Implementation-defined

C.1.2 Annotation Type Definition

```
@Target({TYPE, METHOD})
public @interface HttpGetBinding {
    String location() default "";
}
```

C.1.3 Example

```
@WebService
@HttpGetBinding(location="MyWebServices")
public class MyWebServiceImpl {
    @WebMethod
    @HttpGetBinding(location="ExampleOperation")
    public void myOperation() {
    }
};
```

Appendix D: Change Log

Version 3.0

- Changed specification version and license.

- Changed package name to jakarta.jws.

- Updated Handler Chain Configuration File Schema in Appendix B to accommodate changed namespace.

Version 2.1

- Changed specification version and license information.

- Updated Handler Chain Configuration File Schema in Appendix B to fix errors, add missing elements, and make it valid.

- Updated Section 3.1 to clarify the explanation of when an SEI is implicitly defined by an implementation bean. The revised text refers to the appropriate section 3.3 in the Jakarta XML Web Services 2.0 specification for the actual SEI rules.

- Updated the table in Section 4.4.1 describing the default "name" member value of the jakarta.jws.WebParam annotation to resolve the issue reported. The requirement to specify a name for "document" style, parameter style "BARE", and mode "OUT" or "INOUT" is lifted and default rules are defined. This change is consistent with section 3.6.2.2 of the Jakarta XML Web Services 2.0 specification.

Version 2.0

- Added @WebResult.header.

- A document "bare" style operation can have a void return type and a Holder as a parameter. The Holder of course would have to be INOUT. It could also have 2 parameters one IN and one OUT.

- Changed @SOAPBinding to be configurable on a per operation basis rather than on the entire interface.

- Made @HandlerChain.name and @SOAPMessageHandlers deprecated.

- Added support for Jakarta XML Web Services.

- Updated section 3.1 Service Implementation Bean, Item 6 to state that exposing all public methods if not @WebMethod annotations are declared to include consideration of annotation inheritance.

- Changed default of @WebResult.name to @WebMethod.operationName + "Response" for Doc/lit/bare operations.

- Clarified @WebService.targetNamespace usage

- Clarified @WebParam.mode usage.

- Added @WebMethod.exclude.

- Explicitly stated that if an implementation bean references an endpoint interface, it must implement all the methods in the service endpoint interface.

- Changed @WebParam.name to default to arg0, arg1, etc (based on position in the method signature).

- Added @WebParam.partName and @WebResult.partName to specify part name used in the binding.

- Added requirement that the name for headers must be unique with an operation.

- Clairified that a target namespace of "" maps to the empty namespace, not the web service namespace.

- For doc/lit bare, require that any INOUT or OUT parameters must have a @WebParam.name specified to avoid name clashes with the input parameter.

- Added @WebService.portName for the wsdl:port

- Updated Handler schema

- Added document/literal example to Using Jakarta Web Services Metadata annotation to affect the shape of the WSDL

- Included various editorial changes.

Version 0.9.4

- Allowing @HandlerChain and @SOAPMessageHandler on implementation when an endpointInterface is used.

Version 0.9.3

- Using RFC 2119 Keyword convention.
- Added Retention annotation to spec annotation definitions.
- Fixed various Java and XML syntax errors.
- Changed Implementation Bean to expose all public method by default.
- WSDL generation is REQUIRED.
- Clarified support for Start with WSDL, and Start with WSDL and Java development modes as OPTIONAL.
- Clarified @HandlerChain.file attribute syntax and processing requirements.

Version 0.9.2

- Removed security annotations as these will be defined by JSR-250 – Common Annotations.

Version 0.9.1

- Changed default name of @WebResult to be "return" instead of "result".

- Fixed various Java and XML syntax errors.