Web Services Metadata for the Java^{TM} Platform

Technical Comments to: jsr-181-comments@jcp.org

JSR-181

Java Community Process (JCP)

Version 1.0 June 1, 2005

Specification Lead: Brian Zotter of BEA Systems

Table of Contents

1 Introduction

This specification defines a simplified programming model that facilitates and accelerates the development of enterprise Web Services. J2EE standard deployment technologies, APIs, and protocols require the J2EE developer to master a substantial amount of information. This JSR reduces the amount of information required to implement Web Services on J2EE 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. While the metadata is human readable and editable using a simple text editor, graphical development tools can represent and edit the Java source file using higher levels of abstraction specific to Web Services. This is a simpler and more powerful development environment than traditional coding tools that are used to develop source code using low level APIs.

This specification relies on the JSR-175 specification "A Program Annotation Facility for the Java^{TM} 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.

JSR-181 defines the syntax and semantics of Web Services 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 J2SE 5.0 compiler is assumed. In particular, JSR-181 expects features such as JAX-RPC 1.1 and JSR-109, along with the compiler and language extensions from JSR-175 to be present.

A JSR-181 implementation MUST produce a deployable Java Web Service 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 JSR-181 processors starting from the same valid annotated Java Web Services file MUST produce equivalent Web Service applications, even though they may deploy in very different Java environments. This consistency ensures portability of JSR-181 compliant Java files.

1.1 Expert Group Members

The following people have been part of the JSR-181 Expert Group

Alexander Aptus (Togethersoft Corporation)
John Bossons
Charles Campbell
Alan Davies (SeeBeyond Technology Corp)
Ted Farrell (Oracle)
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 and Jim Trezzo 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 J2EE 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 is possible to write with existing specifications. However, it is a goal to make it easy to build the commonly needed types of Web Services.

2 Concepts

This section provides an overview of the following basic components and processes of the JSR-181 specification:

- Programming model for JSR-181 Web Services.
- Use of metadata in JSR-181.
- Non-normative processing model for a Web Service file.
- Runtime requirements for a JSR-181 container
- Annotations used for WSDL, binding and configuration.

The metadata is formally described in section 4.

2.1 Programming Model Overview

JSR-181 builds on the existing J2EE 1.4 server programming model described by JAX-RPC and JSR-109. A developer who builds a Web Service using 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. JSR-181 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

JSR-181 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

In the “start from Java” development mode, the developer begins by writing a Java class to expose as a Web Service. The developer then runs this Java class through the JSR-181 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 JAX-RPC 1.1. 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.

JSR-181 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

In the “start from WSDL” development mode, the developer uses JSR-181 to implement a pre-defined 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 representing the Java contract, along with Java classes representing 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, JSR-181 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

In the “start with WSDL and Java” development mode, the developer uses JSR-181 annotations to associate a service implementation bean with an existing WSDL contract. In this model, the JSR-181 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 pre-defined `wsdl:operation`. A JSR-181 implementation supporting 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 “JSR-181 processor” is used to denote the code that processes the annotations in a JSR-181 Web Service 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. For example, a JSR-181 processor targeting the J2EE 1.4 EJB container might generate the Home, Remote, and Service Endpoint interfaces associated with the Web Service.

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 JSR-181 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 considered conformant with JSR-181 as long as it produces a Web Service with the proper runtime behavior.

2.4 Runtime Responsibilities

The runtime environment is responsible for providing 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 JSR-181 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:

a. Automatic deployment to a server directory – This is a “drag and drop” deployment model, similar to that used by JSPs. The annotated Web Service 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).

b. Automatic deployment with external overrides – Similar to approach a), 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.

c. Generation of J2EE 1.4 Web Services - In this model, a tool uses the metadata in the annotated Java class to generate a J2EE 1.4 Web Service based on JSR-109 and JAX-RPC 1.1. While the initial Web Service would be generated from the annotated Java source, the result could be further customized through standard deployment tools including JSR-88 deployment plans. This allows for customization of externally modifiable properties at deployment or runtime, without requiring access to the source file for modification and recompilation.

Chapter 9 defines one such a mapping, to the J2EE 1.4 environment. Subsequent revisions to the specification will update this mapping for other environments, such as J2EE 5.0.

2.5 Metadata Use

The metadata used to annotate the service implementation bean conforms to the JSR-175 specification and the specific JSR-181 *annotation type* declarations that are defined in this specification (using the JSR-175 metadata facility). These *annotation type* declarations are contained in packages that need to be imported by every JSR-181 Web Service source file. JSR-175 provides the syntax for expressing the attribute declarations that are in these packages. This JSR specifies the contents of the `javax.jws` and `javax.jws.soap` packages (see attached APIs). These *annotation type* declarations provide a language for expressing Web Service metadata using the JSR-175 metadata facility.

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 `javax.jws` and `javax.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 will also be accessible using 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 JSR-181 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 JSR-181 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 JSR-181 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

JSR-181 defines appropriate defaults for most annotation members. This allows the Java Web Service author to avoid 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. On seeing this marker constant, the JSR-181 processor treats the member-value as though it had the computed default described in this document. For example, when the JSR-181 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

JSR-181 metadata 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. The following example illustrates this, using JSR-175 syntax and the *annotation type* declarations defined in the `javax.jws` and `javax.jws.soap` packages:

```
import javax.jws.WebService;
import javax.jws.WebMethod;

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

JSR-181 includes a full set of metadata tags to define the Web Services that are provided by a service implementation bean. Most of these metadata tags have reasonable defaults, which are explicitly called out in this document. The Java Web Service author can avoid providing tags for the most common Web Service definitions.

The following sections give an overview of the types of annotations provided by JSR-181.

2.6.1 WSDL Mapping Annotations

WSDL mapping annotations control the mapping from Java source onto WSDL constructs. As described in *2.2 Development Models*, this specification supports both a "start from Java" and a "start from WSDL" development mode. When starting from Java, the WSDL mapping annotations control the shape of the WSDL generated from the Java source. When starting from 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.

JSR-181 defines a single set of annotations that map the implementation object to the SOAP protocol binding. JSR-181 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 service's business methods.

3 Server Programming Model

This section describes the server programming model for JSR-181. The JSR-181 server programming model is a simplification of the existing J2EE Web Services server programming models, as defined in JAX-RPC and JSR-109. JSR-181 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 implementing Web Services with JSR-181 is responsible for implementing the service implementation bean containing the Web Service's business logic. A JSR-181 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 4.1Annotation: javax.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, 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 JSR-181 annotations other than `@WebService(endpointInterface` and `serviceName attributes only)`, `@HandlerChain` and `@SOAPMessageHandlers.`. More information on the `@WebService.endpointInterface` attribute may be found in 4.1 Annotation: javax.jws.WebService.
- If the implementation bean does not implement a service endpoint interface, all public methods other than those inherited from java.lang.Object will be exposed as Web Service operations. This behavior can be overridden by using the WebMethod annotation to specify explicitly those public methods that are to be exposed. If a WebMethod annotation is present, only the methods to which it is applied are exposed.

3.2 Service Endpoint Interface

A JSR-181 service implementation bean MAY reference a service endpoint interface, thus separating the contract definition from the implementation. A JSR-181 service endpoint interface MUST meet the requirements specified in JAX-RPC 1.1 [5], section 5.2, 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 of the methods on the service endpoint interface 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 JSR-181 annotations to control the mapping from Java to WSDL. Section 7.1 contains the complete list of annotations allowed on a service endpoint interface.

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 *3.1 Service Implementation Bean* or in *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 JAX-RPC 1.1 [5], section 5.
- The method MAY throw `java.rmi.RemoteException`, but is not REQUIRED to do so.

4 Web Services Metadata

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

4.1 Annotation: javax.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	The simple name of the Java class or interface
targetNamespace	The XML namespace used for the WSDL and XML elements generated from this Web Service.	Implementation-defined, as described in JAX-RPC 1.1 [5], section 5.5.2. Typically derived from the package containing the Web Service.
serviceName	The service name of the Web Service. Used as the name of the `wsdl:service` when mapped to WSDL 1.1. Not allowed on interfaces.	The simple name of the Java class + “Service"
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 JSR-181 tool MUST instead 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 JSR-181 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. Not allowed on 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 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: javax.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.	The name of the Java method
Action	The action for this operation. For SOAP bindings, this determines the value of the `SOAPAction` header	""

4.2.2 Annotation Type Definition

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

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

4.3 Annotation: javax.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 JSR-181 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: javax.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 as it appears in the WSDL. For RPC bindings, this is name of the `wsdl:part` representing the parameter. For document bindings, this is the local name of the XML element representing the parameter.	Name of the parameter as it appears in the argument list.
targetNamespace	The XML namespace for the parameter. Only used with document bindings, where the parameter maps to an XML element.	The targetNamespace for the Web Service
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 JAX-RPC definition of Holder types. See JAX-RPC 1.1 [5], section 4.3.5. OUT and INOUT modes are only supported for RPC bindings or for parameters that map to headers.	IN
header	If true, the parameter is pulled from a message header rather then the message body.	false

4.4.2 Annotation Type Definition

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

   public enum Mode {
      IN,
      OUT,
      INOUT
   };

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

4.4.3 Example

Java Source:

```
@WebService(targetNamespace="http://www.openuri.org/jsr181/WebParamExam
ple")
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class PingService {

   @WebMethod(operationName = "PingOneWay")
   @Oneway
   public void ping(@WebParam(name="Ping") PingDocument p) {
   }

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

   @WebMethod(operationName = "SecurePing")
   @Oneway
   public void ping(
      @WebParam(name="Ping") PingDocument p,
      @WebParam(name="SecHeader", header=true)
          SecurityHeader secHdr) {
   }
};
```

Resulting WSDL:

```
<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://www.openuri.org/jsr181/WebParamExample"
   xmlns:wsdl="http://www.openuri.org/jsr181/WebParamExample"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   targetNamespace="http://www.openuri.org/jsr181/WebParamExample">

   <types>
      <s:schema elementFormDefault="qualified"

targetNamespace="http://www.openuri.org/jsr181/WebParamExample">
         <s:complexType name="Ping">
           . . .
         </s:complexType>

         <s:complexType name="SecurityHeader">
            . . .
         </s:complexType>
         <s:element name="SecHeader" type="SecurityHeader"/>
      </s:schema>
   </ types>

   <message name="PingOneWay">
        <part name="Ping" type="tns:Ping"/>
   </message>

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

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

   <message name="SecurePing">
       <part name="Ping" type="tns:Ping"/>
       <part name="SecHeader" 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="Ping" use="literal"/>
        </input>

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

      </operation>
      <operation name="SecurePing">
```

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

4.5 Annotation: javax.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 as it appears in the WSDL and messages on the wire. For RPC bindings, this is the name of the `wsdl:part` representing the return value. For document bindings, this is the local name of the XML element representing the return value.	“return”
targetNamespace	The XML namespace for the return value. Only used with document bindings, where the return value maps to an XML element.	The targetNamespace for the Web Service.

4.5.2 Annotation Type Definition

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

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: javax.jws.HandlerChain

4.6.1 Description

The `@HandlerChain` annotation associates the Web Service with an externally defined handler chain. This annotation is typically used in scenarios where embedding the handler configuration directly in the Java source is not appropriate; for example, where the handler configuration needs to be shared across multiple Web Services, or where the handler chain consists of handlers for multiple transports.

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 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
name	Name of the handler chain in the configuration file	None

4.6.2 Annotation Type Definition

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

4.6.3 Examples

Example 1

Java Source:
Located in /home/mywork/src/com/jsr181/examples/

package com.jsr181.examples

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

Handler Chain Configuration File
Located in /home/mywork/src/com/jsr181/examples/config/

```
<handler-config>
    <handler-chain>
         <handler-chain-name>StandardHandlerChain</handler-chain-name>
         <handler>
            <handler-name>
               com.fabrikam.handlers.LogHandler
            </handler-name>
            <handler-class>
               com.fabrikam.handlers.LogHandler
            </handler-class>
            <init-param>
               <param-name>logCategory</param-name>
               <param-value>MyService</param-value>
            </init-param>
         </handler>
         <handler>
            <handler-name>
               com.fabrikam.handlers.AuthorizationHandler
            </handler-name>
            <handler-class>
              com.fabrikam.handlers.AuthorizationHandler
            </handler-class>
            <soap-role>SecurityProvider</soap-role>
            <soap-header
               xmlns:ns2=http://openuri.org/examples/security>
                ns2:SecurityHeader
            </soap-header>
          </handler>
         <handler>
            <handler-name>
               com.fabrikam.handlers.RoutingHandler
            </handler-name>
            <handler-class>
               com.fabrikam.handlers.RoutingHandler
            </handler-class>
         </handler>
   </handler-chain>
</handler-config>
```

Example 2

package com.jsr181.examples

```
@WebService
@HandlerChain(
   file="http://myhandlers.foo.com/ProjectHandlers.xml",
   name="StandardHandlerChain")
public class MyWebService {
};
```

4.7 *Annotation: javax.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 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})
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
@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/jsr181/SoapBindingExample1"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   targetNamespace="http://www.openuri.org/jsr181/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/jsr181/SoapBindingExample1/concat"/>
```

```
         <input>
            <soap:body parts="first second third" use="literal"/>
         </input>
         <output>
            <soap:body parts="return" use="literal"/>
         </output>
   </binding>
</definitions>
```

Example 2 – DOCUMENT/LITERAL/BARE

Java source:

```
@WebService
@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class DocBareService {

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

Resulting WSDL:

```
<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://www.openuri.org/jsr181/SoapBindingExample2"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   targetNamespace="http://www.openuri.org/jsr181/SoapBindingExample2">

   <types>
     <s:schema elementFormDefault="qualified"
targetNamespace="http://www.openuri.org/jsr181/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" type="tns:SubmitPOResponse"/>
   </message>

   <portType name="DocBareService">
```

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

    <binding name="DocBareServiceHttpSoap" type="ExampleService">
       <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
       <operation name="SubmitPO">
          <soap:operation
soapAction="http://www.openuri.org/jsr181/SoapBindingExample2/SubmitPO"
/>
          <input>
             <soap:body parts="parameters" use="literal"/>
          </input>
          <output>
             <soap:body parts="parameters" use="literal"/>
          </output>
    </binding>
</definitions>
```

Example 3 – DOCUMENT/LITERAL/WRAPPED

Java source:

```
@WebService
@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:

```
<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://www.openuri.org/jsr181/SoapBindingExample3"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   targetNamespace="http://www.openuri.org/jsr181/
SoapBindingExample3">

   <types>
     <s:schema elementFormDefault="qualified"

targetNamespace="http://www.openuri.org/jsr181/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/jsr181/SoapBindingExample3/SubmitPO"
/>
         <input>
            <soap:body parts="parameters" use="literal"/>
         </input>
         <output>
            <soap:body parts="parameters" use="literal"/>
         </output>
   </binding>
</definitions>
```

4.8 Annotation: javax.jws.soap.SOAPMessageHandlers

4.8.1 Description

Specifies a list of SOAP protocol handlers that run before and after business methods on the Web Service. These handlers are called in response to SOAP messages targeting the service. The handler API and processing rules are described in JAX-RPC 1.1 [5], section 12.

The `@SOAPMessageHandlers` annotation is an array of `SOAPMessageHandler` types. The handlers are run in the order in which they appear in the annotation, starting with the first handler in the array. Each `SOAPMessageHandler` has the following member-values:

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

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

Member-Value	Meaning	Default
name	Name of the handler.	Name of the handler class
className	Name of the handler class	None
initParams	Array of name/value pairs that will be passed to the handler during initialization	No init params
roles	List of SOAP roles implemented by the handler	No roles
headers	List of SOAP headers processed by the handler. Each element in this array contains a QName which defines the header element processed by the handler. The QNames are specified using the string notation described in the documentation for javax.xml.namespace.QName.valueOf(String qNameAsString).	No headers

4.8.2 Annotation Type Definition

```
public @interface InitParam {
    String name();
    String value();
};

public @interface SOAPMessageHandler {
    String name() default "";
    String className();
    InitParam[] initParams() default {};
    String[] roles() default {};
    String[] headers() default {};
};

@Retention(value=RetentionPolicy.RUNTIME)
Target({TYPE})
public @interface SOAPMessageHandlers {
    SOAPMessageHandler[] value();
};
```

4.8.3 Example:

Java source (simple scenario):

```
@WebService
@SOAPMessageHandlers({
   @SOAPMessageHandler(
       className = "com.fabrikam.handlers.LogHandler"),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.AuthorizationHandler"),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.RoutingHandler")
})
public class MyWebService {

   @WebMethod
   public String echo(String input) {
      return input;
   }
};
```

Java source (complex scenario):

```
@SOAPMessageHandlers({
   @SOAPMessageHandler(
          className = "com.fabrikam.handlers.LogHandler",
          initParams = {@InitParam(name="logCategory",
value="MyService")}
    ),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.AuthorizationHandler",
       roles = {"SecurityProvider"},
       headers =
{"{http://openuri.org/examples/security}/SecurityHeader"}
    ),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.RoutingHandler"
    )
})
public class MyWebService {

   @WebMethod
   public String echo(String input) {
      return input;
   }
};
```

5 Java Mapping To XML/WSDL

One of the main purposes of JSR-181 is to influence the shape of WSDL generated from a Java Web Service. This section defines the mapping from Java to XML/WSDL. By default, JSR-181 follows the Java to XML/WSDL mapping defined in JAX-RPC 1.1 [5] section 5, 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 JAXB or SDO (JSR-235). Annotations for such extensions are out-of-scope for this specification.

5.1 Service Endpoint Interface

JAX-RPC defines a service endpoint interface as the Java representation of an abstract WSDL contract. A service endpoint interface MAY include the following JSR-181 annotations to customize its mapping to WSDL:

- `@WebService.name`, `@WebService.targetNamespace`, and `@WebService.wsdlLocation`
- `@WebMethod` (all attributes)
- `@Oneway`
- `@WebParam` (all attributes)
- `@WebResult` (all attributes)
- `@SOAPBinding` (all attributes)

As described in JAX-RPC 1.1 [5] section 5.2, 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` attributes, 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 names of the `wsdl:port` and `wsdl:binding` sections are not significant and are left implementation-defined.

5.3 Web Method Mapping

Each exposed `web method` in a JSR-181 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 present. If not present, the `wsdl:operation` local name is mapped from the name of the Java method according to the rules defined in JAX-RPC 1.1 [5], section 5.5.5.

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 JAX-RPC 1.1 [5], section 5.5.5.

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 JSR-181 implementation supports bindings other than SOAP 1.1, it MUST include a mechanism to selectively enable or disable these bindings.

By default JSR-181 follows the SOAP binding defined in JAX-RPC 1.1 [5], section 6. The `@SOAPBinding`, `@WebParam`, and `@WebResult` annotations allow the developer to further customize this binding for a particular Web Service.

6.1 Operation Modes

JSR-181 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 names of the parts default to the names of the parameters. If a parameter is annotated with `@WebParam.name` annotation, this name will be used as the name of the part instead of the parameter name. 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. JSR-181 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.

6.1.2.1 Document "Wrapped" Style

In the "wrapped" operation style, the input and output messages contain a single part which refers (via 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 return value, if any, is the sole child element of the global element declaration for the output message.

The `@WebParam.name` and `@WebParam.targetNamespace` attributes determine the QName of a parameter's child element, while the `@WebResult.name` and `@WebResult.targetNamespace` annotations determines the local name 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.*

6.1.2.2 Document "Bare" Style

In the "bare" operation style, the input and output messages contain a single part which refers (via 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 Qame 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 several restrictions. Each document "bare" operation MUST have only a single non-header parameter. If the operation is not marked `@Oneway`, it MUST also have a non-void return value. Further, 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` attribute 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` attribute. 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.

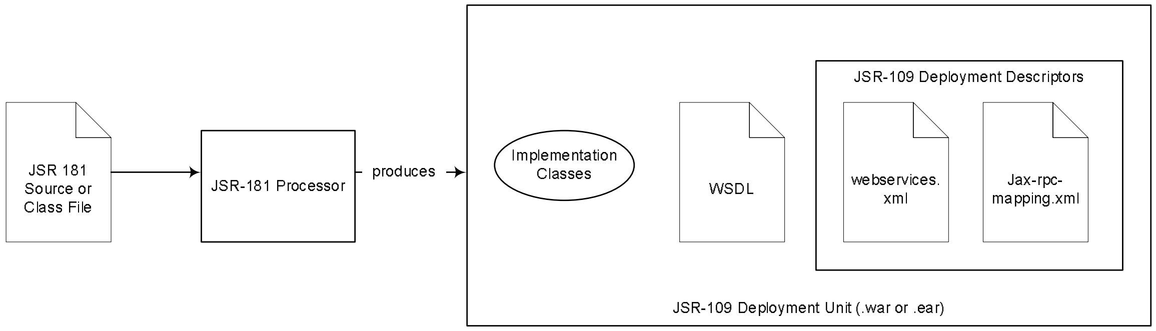
7 Mapping JSR-181 to the J2EE 1.4 Runtime Environment

7.1 Overview

This section describes the mapping of JSR-181 metadata onto J2EE 1.4 Web Services, as defined in JAX-RPC 1.1 and JSR-109 v1.1.

Note: Although this specification defines a standard mapping for metadata to JSR-109 deployment descriptors, the actual production or export of JSR-109 artifacts is OPTIONAL. Semantic support for this mapping is mandatory and is what guarantees portable deployment across J2EE environments.

The following diagram shows the non-normative processing model for JSR-181 on top of J2EE 1.4.


JSR 181 Source or Class File
JSR-181 Processor
produces
Implementation Classes
WSDL
JSR-109 Deployment Descriptors
webservices. xml
Jax-rpc-mapping.xml
JSR-109 Deployment Unit (.war or .ear)

One or more service implementation beans are passed to the JSR-181 processor in source or class form. The JSR-181 processor examines the annotations in each Web Service file and uses this information to generate a JSR-109 deployment unit (EAR or WAR file). The JSR-109 deployment unit contains the following artifacts:

- The compiled Java classes required for deployment. This includes the Web Service implementations, all referenced Java classes, and any generated Home or Remote interfaces.
- WSDL documents describing each service. If `@WebService.wsdlLocation` is specified, the WSDL file will be copied with all of its dependencies from this location and placed into the deployment unit. If `@WebService.wsdlLocation` is not specified, the WSDL file is generated from the annotations in the service implementation bean.
- Service endpoint interfaces defining the methods implemented by each service. The service endpoint interface may be written by the developer or generated from the annotations on the service implementation bean.
- The `webservices.xml` deployment descriptor. This deployment descriptor contains references to all of the Web Services in the deployment unit. There will

be one `webservice` entry for each Web Service class processed by the JSR-181 processor. Section *7.4.1 Mapping from JSR-181 to webservices.xml* describes the mapping from JSR-181 annotations to entries in `webservices.xml`.

- The `jax-rpc-mapping.xml` deployment descriptor. Section 7.4.3 - Mapping from JSR-181 to jax-rpc-mapping.xml describes the mapping from JSR-181 annotations to entries in `jax-rpc-mapping.xml`.
- One of either a `web.xml` or `ejb-jar.xml` deployment descriptor. The JSR-181 processor will generate either a `web.xml` deployment descriptor (for servlet endpoints) or an `ejb-jar.xml` deployment descriptor (for stateless session beans) depending on the implementation model chosen for the service. See the next section for more information on selecting an implementation model.

Once generated, the values in the deployment descriptors may be overridden by the deployer or administrator using standard deployment tools, including JSR 88 deployment plans. This capability allows customization of the Web Service without requiring access to the source code.

7.2 Implementation Models

J2EE 1.4 Web Services may be implemented as plain classes in the servlet container or as stateless session beans in the EJB container. A JSR-181 implementation targeting J2EE 1.4 MUST provide an out-of-band mechanism allowing the developer to specify explicitly the container in which a particular Web Service implementation should run. Future versions of the specification will combine with other J2EE 5.0 technologies (such as EJB3) to provide a standard annotation-driven approach to specifying container-specific technologies and qualities of service.

7.3 Service Endpoint Interface

Every J2EE 1.4 Web Service is required to reference a service endpoint interface that defines the methods implemented by the service. If the JSR-181 service implementation bean does not reference a service endpoint interface through the `@WebService.endpointInterface` annotation, the JSR-181 processor is REQUIRED to generate the interface from the annotations on the implementation bean. The generated service endpoint interface is given an implementation-defined name, placed in an implementation-defined package, and contains a method for each designated `web method` on the service implementation bean. The signature of each service endpoint interface method is identical to that of the corresponding `web method`, except that it includes `java.rmi.RemoteException` in the list of checked exceptions.

7.4 Mapping from JSR-181 to JSR-109 Deployment Descriptors

The JSR-109 deployment model for Web Services defines two deployment descriptors - the JSR-109 deployment descriptor (`webservices.xml`) and the JAX-RPC mapping file (`jax-rpc-mapping.xml`). These files contain deployment and mapping information for J2EE Web Services. The following sections describe the mapping from JSR-181 to the JSR-109 deployment descriptors. A JSR-181 implementation that produces JSR-109 deployment descriptors MUST conform to this mapping.

7.4.1 Mapping from JSR-181 to webservices.xml

Every endpoint supported by a JSR-181 Web Service results in a unique `webservice` entry in `webservices.xml`. The following table shows the relationship between elements in the `webservice` and JSR-181 annotations:

webservices.xml entry	JSR-181 Annotation and Member-Value
webservices/webservice	One entry per Web Service
webservices/webservice/wsdl-file	WSDL generated from service implementation bean or copied from `@WebService.wsdlLocation`
webservices/webservice/jax-rpc-mapping-file	JAX-RPC mapping file generated from service implementation bean
webservices/webservice/port-component/port-component-name	Implementation defined
webservices/webservice/port-component/wsdl-port	Implementation defined
webservices/webservice/port-component/service-endpoint-interface	Generated from the annotations defined on the Web Service
webservices/webservice/service-impl-bean/ejb-link or webservices/webservice/service-impl-bean/service-link	Implementation defined
webservices/webservice/handler	`@SOAPMessageHandlers` or `@HandlerChain`

7.4.2 Example – Mapping JSR-181 to webservices.xml

Java Source:

```
@WebService(targetNamespace="http://www.openuri.org/MyNamespace"
            name="StockQuoteService")
@SOAPBinding
@SOAPMessageHandlers({
   @SOAPMessageHandler(
       className = "com.fabrikam.handlers.LogHandler",
       initParams = {
          @InitParam(name="logCategory", value="MyService")
       }
    ),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.AuthorizationHandler"
       roles = {"SecurityProvider"},
       headers = {
          "{http://openuri.org/examples/security}SecurityHeader"
       }
    ),
    @SOAPMessageHandler(
       className = "com.fabrikam.handlers.RoutingHandler"
    )
})
public class StockQuoteServiceImpl {

     @WebMethod(operationName="GetLastTrade")
     public long getLastTrade(String tickerSymbol) {
        // . . .
     }

    @WebMethod(operationName="GetTickerSymbols")
    public String[] getTickerSymbols() {
        // . . .
    }
}
```

Resulting webservices.xml:

```
<webservices>
    <webservice
         xmlns:ns1="http://www.openuri.org/MyNamespace">
      <wsdl-file>StockQuoteService.wsdl</wsdl-file>
      <port-component>
         <port-component-name>
             StockQuoteServiceSoapHttp
         </port-component-name>
         <service-endpoint-interface>
             StockQuoteService
         </service-endpoint-interface>
         <wsdl-port>ns1:StockQuoteServiceSoapHttpPort</wsdl-port>

         <handler>
            <handler-name>
               com.fabrikam.handlers.LogHandler
            </handler-name>
            <handler-class>
               com.fabrikam.handlers.LogHandler
            </handler-class>
            <init-param>
               <param-name>logCategory</param-name>
               <param-value>MyService</param-value>
            </init-param>
         </handler>
         <handler>
            <handler-name>
               com.fabrikam.handlers.AuthorizationHandler
            </handler-name>
            <handler-class>
              com.fabrikam.handlers.AuthorizationHandler
            </handler-class>
            <soap-role>SecurityProvider</soap-role>
            <soap-header
               xmlns:ns2=http://openuri.org/examples/security>
                ns2:SecurityHeader
            </soap-header>
          </handler>
         <handler>
            <handler-name>
               com.fabrikam.handlers.RoutingHandler
            </handler-name>
            <handler-class>
               com.fabrikam.handlers.RoutingHandler
            </handler-class>
         </handler>
         <ejb-link>StockQuoteServiceEJB</ejb-link>
       </port-component>
    </webservice>
</webservices>
```

7.4.3 Mapping from JSR-181 to jax-rpc-mapping.xml

Every JSR-181 Web Service results in a unique `service-endpoint-mapping` entry in the `jax-rpc-mapping` file. The following table shows the relationship between the elements in the `service-endpoint-mapping` and JSR-181 annotations.

Jax-rpc-mapping.xml entry	JSR-181 Annotation and Member-Value
/java-wsdl-mapping/service-endpoint-mapping	One entry per Web Service
/java-wsdl-mapping/service-endpoint-mapping/service-endpoint-interface	Generated from the annotations on the Web Service.
/java-wsdl-mapping/service-endpoint-mapping/wsdl-port-type	`@WebService.name`
/java-wsdl-mapping/service-endpoint-mapping/wsdl-binding	Implementation defined
/java-wsdl-mapping/service-endpoint-mapping/service-endpoint-method-mapping/wsdl-operation	`@WebMethod.operationName`
/java-wsdl-mapping/service-endpoint-mapping/service-endpoint-method-mapping/wrapped-element	Present if `@SOAPBinding.parameterStyle` is `WRAPPED`

7.4.4 Example – Mapping from JSR-181 to jax-rpc-mapping.xml

Java Source:

```
@WebService(targetNamespace="http://www.openuri.org/MyNamespace"
            name="StockQuoteService")
@SOAPBinding(name="StockQuoteServiceSoap")
public class StockQuoteServiceImpl {

     @WebMethod(operationName="GetLastTrade")
     public GetLastTradeResponse getLastTrade(
            GetLastTradeRequest request) {
        // . . .
     }

    @WebMethod(operationName="GetTickerSymbols")
    public String[] getTickerSymbols() {
        // . . .
    }
}
```

Resulting java-xml-rpc-mapping.xml:

```
<java-wsdl-mapping>
    <service-endpoint-mapping
        xmlns:ns1="http://www.openuri.org/MyNamespace">
         <service-endpoint-interface>StockQuoteService</service-
endpoint-interface>
         <wsdl-port-type>ns1:StockQuoteService</wsdl-port-type>
         <wsdl-binding>ns1:StockQuoteServiceSoap</wsdl-binding>
         <service-endpoint-method-mapping>
              <java-method-name>getLastTrade</java-method-name>
              <wsdl-operation>GetLastTrade</wsdl-operation>
              <wrapped-element/>
         </service-endpoint-method-mapping>
         <service-endpoint-method-mapping>
              <java-method-name>getTickerSymbols</java-method-name>
              <wsdl-operation>GetTickerSymbols</wsdl-operation>
              <wrapped-element/>
         </service-endpoint-method-mapping>
    </service-endpoint-mapping>
</java-wsdl-mapping>
```

8 Using JSR-181 annotations to affect the shape of the WSDL

8.1 RPC Literal Style

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

Java source:

```
import javax.jws.*;
import javax.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
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:tns="http://openuri.org/11/2003/ExampleWebService"
   xmlns:wsdl="http://openuri.org/11/2003/ExampleWebService"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
targetNamespace="http://openuri.org/11/2003/ExampleWebService">
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <types>
   <s:schema elementFormDefault="qualified"
     targetNamespace="http://openuri.org/11/2003/ExampleWebService">

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

     <s:element name="Customer" type="Customer"/>

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

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

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

     <s:element name="notifyTransfer">
       <s:complexType>
          <s:sequence>
            <s:element name="CustomerId" type="s:string"
                         minOccurs="1" maxOccurs="1"/>
            <s:element name="TransferData" type="TransferDocument"
                         minOccurs="1" maxOccurs="1"/>
          </s:sequence>
       </s:complexType>
     </s:element>
</s:schema>
 </types>

 <message name="notifyTransfer">
   <part name="parameters" element="NotifyTransfer"/>
 </message>

 <message name="notifyTransferHeaders"”>
   <part name="Token" element="Token"/>
 </message>

 <message name="createCustomer">
   <part name="Customer" type="Customer"/>
 </message>

 <message name="createCustomerHeaders">
   <part name="Token" type="Token"/>
 </message>

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

 <message name="login">
   <part name="UserName" type="s:string"/>
```

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

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

    <portType name="ExampleWebService">
        <operation name="login">
     <input message="tns:login"/>
           <output message=”tns:loginResponse”/>
        </operation>

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

        <operation name="createCustomer">
            <input message="tns:createCustomer"/>
            <output message="tns:createCustomerResponse"/>
        </operation>
    </portType>

    <binding name="ExampleWebServiceSoapHttp" type="ExampleWebService">
       <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                     style="document"/>
       <operation name="login">
           <soap:operation soapAction="urn:login"/>
           <input>
             <soap:body parts="UserName Password" use="literal"/>
           </input>
           <output>
             <soap:body parts="Token" use="literal"/>
           </output>
       </operation>
       <operation name="createCustomer">
           <soap:operation soapAction="urn:createCustomer"/>
           <input>
             <soap:body parts="Customer" use="literal"/>
             <soap:header message="createCustomerHeaders"
                  part="Token" use="literal"/>
           </input>
           <output>
             <soap:body parts="CustomerId" use="literal"/>
           </output>
       </operation>
       <operation name="notifyTransfer">
           <soap:operation soapAction="urn:notifyTransfer"
                           style="document"/>
           <input>
             <soap:body parts="parameters" use="literal"/>
             <soap:header message="notifyTransferHeaders"
                  part="Token" use="literal"/>
           </input>
       </operation>
    </binding>
</definitions>
```

9 References

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

2. JSR-88 J2EE Application Deployment
   http://jcp.org/en/jsr/detail?id=88

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

4. J2EE 1.4
   http://jcp.org/en/jsr/detail?id=151

5. JAX-RPC V1.1
   http://www.jcp.org/en/jsr/detail?id=101

6. Implementing Enterprise Web Services 1.1 (was JSR-109)
   http://www.jcp.org/en/jsr/detail?id=921

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

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

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

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

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

Appendix A: Relationship to Other Standards

JSR-181 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: JSR-181 does not define a container or runtime environment – implementers provide tools to map the Java classes to specific runtime environments. The functionality of the J2EE 1.4 containers is assumed. The features provided by JAX-RPC 1.1 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 JSR-109 deployment descriptors is provided in JSR-181.

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 JSR-181.

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

Appendix B: Handler Chain Configuration File Schema

This is the schema for the handler configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://www.bea.com/xml/ns/jws"
          xmlns:jws="http://www.bea.com/xml/ns/jws"
          xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          elementFormDefault="qualified"
          attributeFormDefault="unqualified"
          version="1.1">

  <xsd:annotation>
    <xsd:documentation>
     <![CDATA[

      This is the schema definition for the handler chain configuration
      file used by JSR-181.  It relies on the handler definitions that
      are part of the standard J2EE deployment descriptors.

      ]]>
    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="j2ee_1_4.xsd"/>
  <xsd:include schemaLocation="j2ee_web_services_1_1.xsd"/>

  <xsd:complexType name="handler-chainType">
    <xsd:sequence>
      <xsd:element name="handler-chain-name"
               type="j2ee:string"/>
      <xsd:element name="handler"
               type="j2ee:port-component_handlerType"
               minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="handler-configType">
    <xsd:sequence>
      <xsd:element name="handler-chain"
                   type="jws:handler-chainType"
                   minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="handler-config" type="jws:handler-configType"/>
</xsd:schema>
```

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 0.9.1

- Changed default name of @WebResult to be "return" instead of "result".
- Fixed various Java and XML syntax errors.

Version 0.9.2

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

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 0.9.4

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