JAX-WS Developer's Guide

JOnAS Team ()

- March 2009 -

Table of Contents

List of Tables

Preface

JAX-WS stands for *J*ava *A*PI for *X*ML *W*eb*S*ervices. It's a technology used to implement webservices endpoints and webservices clients communicating using XML.

SOAP and XML messaging is a complex domain, but JAX-WS aims to hide the complexity of that domain. Endpoint development is an easy task: the developer writes a Java interface that will defines the available webservices operations, writes a concrete class (can be multiple class in case of inheritance) that will implement the Java interface. A minimal set of annotations is required to declares the class and interface as webservices. Writing webservices client is also an easy task: using generated classes from a given WSDL, the client can access the webservice without knowing any technical details. This is the role of the JAX-WS engine to do all the XML marshalling/unmarshalling, SOAP processing, ...

With JAX-WS, developers takes both advantages of a standard Java webservices specification (portability) and of the Java platform independence (OS/hardware neutral). Moreover, with JAX-WS (and webservices in general), a client can access a webservice not implemented with Java and vice versa. This is possible because JAX-WS respects the W3C recommendations (*W*orld *W*ide *W*eb *C*onsortium) for HTTP, SOAP and WSDL (WebServices Description Language).

Chapter 1. Developing a Webservice Endpoint

A Webservice endpoint is the implementation of a webservice. It's a server side artifact that can answer to webservices requests.

An endpoint can be implemented using a POJO or using a Stateless EJB3.

Two approaches can be used to develop an endpoint:

- Starting from Java
- Starting from WSDL

Taking one of theses 2 approaches is the developer's choice.

1.1. Starting from Java

Choosing a Java-first approach is typical of a webservices development from scratch.

1.1.1. Defining the Service Endpoint Interface

The service endpoint interface (SEI) is a Java interface that specified the methods that will be exposed as webservices operations. This interface is required to be annotated with the @WebService annotation.

```
@WebService(name="QuoteReporter",
            targetNamespace="http://jonas.ow2.org/examples/jaxws/quote")
public interface QuoteReporter {
    public Quote getQuote(@WebParam(name="ticker") String ticker);
}
```

The @WebService[1] annotation, placed on a SEI, provides the information required to generate a valid WSDL:

- name: this value will be used as the `wsdl:portType` name in the generated WSDL
- targetNamespace: this value will be used as the XML namespace containing the WSDL.

  It's required that this value is a URI, it's NOT required to use an URL or that this URL is accessible.

Note

Annotating the SEI will provides all the necessary information to build an abstract WSDL (types + messages + portType).

The `@WebParam`[2] annotation is placed on the method parameter because Java compilation do not store the parameter name ('ticker' here) in the byte code. Having a @WebParam ensure that the generated WSDL will always have the correct operation parameter names.

1.1.2. Implementing the Service Endpoint Interface

Once the SEI has been created, the developer has to implement the SEI.

```
@WebService(portName="QuoteReporterPort",
```

[1] http://java.sun.com/javaee/5/docs/api/javax/jws/WebService.html
[2] http://java.sun.com/javaee/5/docs/api/javax/jws/WebParam.html

```
            serviceName="QuoteReporterService",
            targetNamespace="http://jonas.ow2.org/examples/jaxws/quote",
            endpointInterface="org.ow2.jonas.examples.jaxws.quote.QuoteReporter")
public class QuoteReporterBean implements QuoteReporter {
    public Quote getQuote(final String ticker) {
        return new Quote(ticker, Math.random() * 100);
    }
}
```

Once again, the developer has to annotate its class with the @WebService annotation.

- portName: this value specifies the `wsdl:port` to be used
- serviceName: this value specifies the `wsdl:service` to be used
- targetNamespace: this value specifies the targetNamespace of the `wsdl:service`
- endpointInterface: this value specifies the service endpoint interface classname (SEI)

That's it, a first simple webservice endpoint code ! The next section will focus on the most important webservices annotations.

1.1.3. Annotating the code

1.1.4. Generate WSDL

Once you have a codebase fully annotated, you may choose to generate a WSDL contract.

Note

With JOnAS 5.1 M4, there is no simple way to perform a wsdl2java operation, although the Apache CXF ant tasks are available.

Next releases will provide more examples and documentation about the generation process.

1.2. Starting from WSDL

In the WSDL first approach, the developer is required to implement a webservice that is constrained by a WSDL contract.

In that case, 2 options are available:

- Generates, from the WSDL, the SEI interface and an implementation skeleton.
- Implements a Provider that will allow the developer to work at the XML level.

1.2.1. Generates skeleton

JAX-WS tools are provided to help the developers to quickly start to implement business code by generating most of the plumbing. wsdl2java for a webservice endpoint will generate the following artifacts:

- {portType}.java: this is the fully annotated SEI interface. By default, its name is based on the wsdl:portType name.
- {portType}Impl.java: this is an implementation skeleton, annotated with @WebService. The developer just have to fill the blanks in this class.
- {types}.java + ObjectFactory.java: theses JAXB generated classes represents the operations parameters and documents.

Note

Insert here a picture of generated classes.

1.2.1.1. Generates the Service Endpoint Interface

A SEI generated by a JAX-WS 2.1 compatible tool look like this:

1.2.1.2. Implements the Service Endpoint Interface

Implementing the SEI simply means that the developer has to fill in the blanks of the generated skeleton. This part focus on business code.

1.3. Writing a Provider

JAX-WS allows developers to write endpoints dealing directly with XML messages using the Provider interface. This can be done by implementing Provider<Source> or Provider<SOAPMessage> or Provider<DataSource>.

```
T invoke(T request) throws WebServiceException;
```

Supported parameter's types are: Source[3], SOAPMessage[4], DataSource[5]. They define the format of the data that will be manipulated inside the invoke method. What will be available in theses data structures depends on the ServiceMode specified. ServiceMode can be either MESSAGE or PAYLOAD (PAYLOAD being the default when no mode is specified):

- ServiceMode.MESSAGE: The structure will represent the whole request message (allowing access to the SOAP envelop and SOAP headers for example)
- ServiceMode.PAYLOAD: The structure will represent only the body of the request (restricting access to business data located inside the SOAP Body element)

When writing a Provider, the webservice developer MUST starts with a WSDL, because the Provider implementation do not provide any exploitable information about the XML data structures expected (such as bean class parameters and such).

Note

Providers cannot be implemented using Stateless EJB.

Providers implementation supports the one way MEP (Message Exchange Pattern) simply by returning null.

1.3.1. Supported modes

Although many modes and types combinations are possibles, there are some logical restrictions given the Binding type (HTTP, SOAP) in use.

Supported combinations are described below.

1.3.1.1. Using SOAP Binding (1.1 or 1.2)

In the absence of any customization (like applying a @BindingType annotation on the Provider), the binding is assumed to be SOAP 1.1.

[3] http://java.sun.com/j2se/1.5/docs/api/javax/xml/transform/Source.html
[4] http://java.sun.com/javaee/5/docs/api/javax/xml/soap/SOAPMessage.html
[5] http://java.sun.com/javaee/5/docs/api/javax/activation/DataSource.html

Caution

Provider<SOAPMessage> used in conjunction with a ServiceMode set to PAYLOAD is invalid.

Indeed the SOAPMessage type represents the whole message, not just the SOAP Body element.

Caution

Provider<DataSource> is *always invalid* when used with a SOAP Binding.

1.3.1.1.1. Provider<Source> and MESSAGE mode

Using a Provider<Source> with the MESSAGE mode is used to change the message's content (provided as a javax.xml.transform.Source object) using XSLT Transformers.

```
@WebServiceProvider
@ServiceMode(Service.Mode.MESSAGE)
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source message) {
        // Process the request
        ...
        // Prepare the response Source
        Source response = ...;
        return response;
    }
}
```

1.3.1.1.2. Provider<Source> and PAYLOAD mode

Using a Provider<Source> with the PAYLOAD mode is used to change the message's payload (provided as a javax.xml.transform.Source object) using XSLT Transformers.

```
@WebServiceProvider
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source source) {
        // Process the request
        ...
        // Prepare the response Source
        Source response = ...;
        return response;
    }
}
```

1.3.1.1.3. Provider<SOAPMessage> and MESSAGE mode

Using a Provider<SOAPMessage> with the MESSAGE mode is used to change the message's content (provided as a javax.xml.soap.SOAPMessage object) with a DOM-like API.

```
@WebServiceProvider
@ServiceMode(Service.Mode.MESSAGE)
public class ProviderImpl implements Provider<SOAPMessage> {
    public SOAPMessage invoke(SOAPMessage message) {
        // Process the request
        ...
        // Prepare the response SOAPMessage
        SOAPMessage response = ...;
        return response;
    }
}
```

Caution

Provider<SOAPMessage> used in conjunction with a ServiceMode set to PAYLOAD is *invalid*.

Indeed the `SOAPMessage` type represents the whole message, not just the SOAP Body element.

1.3.1.2. Using HTTP Binding

Providers can also be implemented using the HTTP binding. That means that no SOAP messages are involved and that the interaction will only be based on XML/HTTP.

Caution

Provider<DataSource> used in conjunction with a ServiceMode set to PAYLOAD is invalid.

Caution

Provider<SOAPMessage> is *always invalid* when used with an HTTP Binding.

Indeed, using the HTTP binding assume that the messages are not SOAP messages.

1.3.1.2.1. Provider<Source> and MESSAGE mode

Using a Provider<Source> with the MESSAGE mode and the HTTP binding is used to change the message's content (provided as a javax.xml.transform.Source object) using XSLT Transformers.

```
@WebServiceProvider
@ServiceMode(Service.Mode.MESSAGE)
@BindingType(HTTPBinding.HTTP_BINDING)
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source message) {
        // Process the request
        ...
        // Prepare the response Source
        Source response = ...;
        return response;
    }
}
```

1.3.1.2.2. Provider<Source> and PAYLOAD mode

Using a Provider<Source> with the PAYLOAD mode and the HTTP binding is used to change the message's body (provided as a javax.xml.transform.Source object) using XSLT Transformers.

```
@WebServiceProvider
@BindingType(HTTPBinding.HTTP_BINDING)
public class ProviderImpl implements Provider<Source> {
    public Source invoke(Source message) {
        // Process the request
        ...
        // Prepare the response Source
        Source response = ...;
        return response;
    }
}
```

1.3.1.2.3. Provider<DataSource> and MESSAGE mode

Using a Provider<DataSource> with the MESSAGE mode and the HTTP binding is used to process the message attachment (provided as a javax.xml.transform.Source object). This style of provider is useful when dealing *only* with message's attachments (no XML changes).

```
@WebServiceProvider
@ServiceMode(Service.Mode.MESSAGE)
@BindingType(HTTPBinding.HTTP_BINDING)
```

```
public class ProviderImpl implements Provider<DataSource> {
    public DataSource invoke(DataSource message) {
        // Process the request
        ...
        // Prepare the response DataSource
        DataSource response = ...;
        return response;
    }
}
```

1.3.2. Wrap up

The table below summarizes the requirement with all the possible combinations ([6]).

	XML/HTTP Binding	SOAP/HTTP Binding
	Provider<Source>	
Payload	Primary part or content as Source	SOAP Body from the primary part or SOAP Body as Source
Message	Primary part or content as Source	SOAP Envelope from the primary part or SOAP Envelope as Source
Provider<DataSource>		
Payload	Not Valid [1]	Not Valid [1]
Message	DataSource as an object	Not Valid [2]
Provider<SOAPMessage>		
Payload	Not Valid [3]	Not Valid [3]
Message	Not Valid [4]	SOAPMessage as an object

6

Note

Provider<DataSource> is used for sending attachments and thus payload mode is not valid.

Note

Provider<DataSource> in SOAP/HTTP is not valid since attachments in SOAP are sent using Provider<SOAPMessage>.

Note

Provider<SOAPMessage> in payload mode is not valid because the entire SOAPMessage is received, not just the payload which corresponds to the body of the SOAPMessage.

Note

Provider<SOAPMessage> in message mode using XML/HTTP binding is not valid since the client may have sent an XML message that may not be SOAP.

1.3.3. Implementing RESTful services using Providers

[6] Source Arun Gupta (http://weblogs.java.net/blog/arungupta/archive/2006/03/jaxws_20_provid_1.html) [http://weblogs.java.net/blog/arungupta/archive/2006/03/jaxws_20_provid_1.html]

Note

Using HTTP Binding + provider can be used to implement simple RESTful services (in the absence of JAX-RS support).

This topic will be covered in future versions in this documentation.

1.4. Overriding annotations

Some annotations values can be overridden in a Java EE application server using the webservices.xml (located in WEB-INF/ or META-INF/ given the implementor is respectively a POJO or an EJB).

The link between @WebService (or @WebServiceProvider) and the XML <port-component> description is done through the <service-impl-bean> element:

- <ejb-link> with an ejb-name pointing to a Stateless EJB3 annotated with @WebService / @WebServiceProvider.
- <servlet-link> with a servlet-name pointing to a Servlet declaration in the web.xml.

Table 1.1. Overriding @WebService

Annotation	Deployment descriptor element	Comment
@WebService	<webservices>/<webservice-description>/<port-component>	One @WebService per <port-component>
@WebService.wsdlLocation	<webservices>/<webservice-description>/<wsdl-file>	Overrides the WSDL location value for all the <port-component> declared in the <webservice-description>
@WebService.name	<webservices>/<webservice-description>/<port-component>/<port-component-name>	
@WebService.serviceName	<webservices>/<webservice-description>/<port-component>/<wsdl-service>	
@WebService.portName	<webservices>/<webservice-description>/<port-component>/<wsdl-port>	
@WebService.endpointInterface	<webservices>/<webservice-description>/<port-component>/<service-endpoint-interface>	

Table 1.2. Overriding @WebServiceProvider

Annotation	Deployment descriptor element	Comment
@WebServiceProvider	<webservices>/<webservice-description>/<port-component>	One @WebServiceProvider per <port-component>
@WebServiceProvider.wsdlLocation	<webservices>/<webservice-description>/<wsdl-file>	Overrides the WSDL location value for all the <port-

Annotation	Deployment descriptor element	Comment
		component> declared in the <webservice-description>
@WebService.serviceName	<webservices>/<webservice-description>/<port-component>/<wsdl-service>	
@WebService.portName	<webservices>/<webservice-description>/<port-component>/<wsdl-port>	
@WebService.endpointInterface	N/A	It is not required for a Provider to specify a SEI

Table 1.3. Overriding @BindingType

Deployment descriptor	Possible values
<port-component>/<protocol-binding>	Specify the protocol binding used by this port-component. The following alias are supported: • `##SOAP11_HTTP`: SOAP 1.1 Binding • `##SOAP11_HTTP_MTOM`: SOAP 1.1 Binding with MTOM enabled • `##SOAP12_HTTP`: SOAP 1.2 Binding • `##SOAP12_HTTP_MTOM`: SOAP 1.2 Binding with MTOM enabled • `##XML_HTTP`: XML Binding
<port-component>/<enable-mtom>	This element permits to enable/disable the SOAP MTOM/XOP mechanism for the endpoint. This element has higher priority than <protocol-binding>, meaning that if `##SOAP11_HTTP_MTOM` is specified as protocol binding and <enable-mtom> is false, it will be equivalent to a protocol binding set to `##SOAP11_HTTP`

1.5. Endpoint packaging

When the webservices implementation has been finished, code compiled, it's time to take care of the application packaging. There are some simple packaging rules to follow.

1.5.1. Web Application

When the endpoint is implemented as a POJO, the implementation class may be referenced as a servlet in the WEB-INF/web.xml. If it is not declared in the web.xml, it will be auto discovered at deployment time. If it is declared in the web.xml, it must be specified as a new servlet element and the servlet must have an associated servlet-mapping.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">
```

```
    <servlet>
        <servlet-name>{servlet-name}</servlet-name>                              1
        <servlet-class>{implementor-fully-qualified-classname}</servlet-class>  2
    </servlet>

    <servlet-mapping>
        <servlet-name>{servlet-name}</servlet-name>                              3
        <url-pattern>{webservice-endpoint-url-pattern}</url-pattern>             4
    </servlet-mapping>

</web-app>
```

1. A usual servlet name: unique among other servlets.
2. The implementor fully qualified classname (It's not a Servlet, but that's OK, it will be changed during deployment).
3. The same servlet name specified first in the servlet element.
4. The URL pattern that will point to the webservice endpoint.

Caution

If the endpoint is declared in the `WEB-INF/web.xml` but no `servlet-mapping` is provided, the endpoint will not be accessible (No URLs will point to the servlet).

If the WSDL is referenced from the endpoint (using @WebService.wsdlLocation(), @WebServiceProvider.wsdlLocation() or webservices.xml), it has to be provided in the `WEB-INF/wsdl/` directory of the webapp.

1.5.1.1. Default Web URL Mapping

The webservice endpoint contained in the webapp will be available through an URL.

If no servlet-mapping was specified in the web.xml a default mapping will be automatically introduced.

Table 1.4. Automatic Web Endpoints Servlet Mappings Patterns

Condition	Servlet Mapping Pattern
`serviceName` has been provided in the @WebService(s)	/{service-qname-local-part}
no `serviceName` provided	/{simple-implementor-classname}Simple

1.5.2. EjbJar

When the webservice endpoint is implemented as a Stateless EJB

1.5.2.1. Endpoint URL Mapping

As an ejbjar is not a webapp, there is no available web context, or url-pattern that can be used to access the endpoint through HTTP.

A Web context is created and named after the ejbjar's filename (stopping at the first '_' character).

Table 1.5. Automatic EJB Endpoints Servlet Mappings Patterns

Condition	Servlet Mapping Pattern
<endpoint-address> provided for the bean in the easybeans.xml	provided value
`serviceName` has been provided in the @WebService(s)	/{service-qname-local-part}

Condition	Servlet Mapping Pattern
no `serviceName` provided	/{simple-implementor-classname}Simple

1.5.2.2. EasyBeans deployment descriptor customizations

The EasyBeans deployment descriptor provides a mean for customizing some webservices endpoint properties.

Note

To be completed with XML references. (easybeans/webservices/endpoint and easybeans/ejb/session/endpoint-address).

Chapter 2. Developing a Webservice Consumer

A Webservice consumer is a client of a webservice endpoint. It's a client side artifact that perform webservices requests.

With JAX-WS, clients can benefits of asynchronous invocations (more on this later).

A webservice consumer can be any of the following Java EE components: EJB (Session/MessageDriven), Servlet (or any web components such as JSP, JSF, ...) or Application Client. For any of theses components, the creation process is the same.

A consumer typically starts with a WSDL (describing the endpoint's contract), but dynamic invocation (working at the XML level) can also be performed.

2.1. Port

A webservice consummer can be implemented using a Port. A Port is a Java interface that hides to the client all the WS low level mechanisms (masrshalling, unmarshalling, handler chains, ...).

A Port is usually generated from a WSDL.

2.1.1. Generates portable client artifacts

When the client's developer starts with a WSDL, it usually generates the client artifacts (Java classes) using a wsdl2java tool.

Note

With JOnAS 5.1 M4, there is no simple way to perform a wsdl2java operation, although the Apache CXF ant tasks are available.

Next releases will provide more examples and documentation about the generation process.

2.1.1.1. Example of generated Service

2.1.1.2. Example of generated Port

2.1.2. Synchronous consumer

Using synchronous consumer is the easiest way of consuming a webservice: by default, wsdl2java tools generate classes supporting only synchronous operations.

The following code snippet shows the Port source code that a wsdl2java tool generate using an input WSDL file.

```
@WebService(targetNamespace = "http://jonas.ow2.org/samples/jaxws/calculator",
            name = "Calculator")
public interface Calculator {

    @ResponseWrapper(localName = "addResponse",
                     targetNamespace = "http://jonas.ow2.org/samples/jaxws/calculator",
                     className = "org.ow2.jonas.samples.jaxws.calculator.AddResponse")
    @RequestWrapper(localName = "add",
                    targetNamespace = "http://jonas.ow2.org/samples/jaxws/calculator",
```

```
                    className = "org.ow2.jonas.samples.jaxws.calculator.Add")
    @WebResult(name = "return", targetNamespace = "")
    @WebMethod
    public int add(@WebParam(name = "arg0", targetNamespace = "")
                   int arg0,
                   @WebParam(name = "arg1", targetNamespace = "")
                   int arg1);
}
```

2.1.3. Asynchronous consumer

By default, WSDL2Java generation do not generate asynchronous methods for each operation of the wsdl:portType.

To enable the async method generation, a binding customization has to be provided during the WSDL2Java execution.

It can be provided as part of the WSDL file:

```
<portType name="CalculatorPort">
    <operation name="add">
      <input message="tns:addRequest"/>
      <output message="tns:addResponse"/>
    </operation>
</portType>
<binding name="CalculatorBinding" type="tns:CalculatorPort">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="add">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/></input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
</binding>
```

Or as a separate bindings declaration file:

```
<bindings xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
          xmlns="http://java.sun.com/xml/ns/jaxws"
          wsdlLocation="...">

    <bindings node="wsdl:definitions">
        <package name="org.ow2.jonas.ws.async.client"/>
        <enableAsyncMapping>true</enableAsyncMapping>
    </bindings>
</bindings>
```

```
// Synchronous method
public int add(int a, int b);

// Asynchronous polling method
public Response<AddResponse> add(int a, int b);

// Asynchronous callback method
public Future<?> add(int a, int b, AsyncHandler<AddResponse>);
```

2.1.3.1. Asynchronous polling

2.1.3.2. Asynchronous callback

2.2. Dispatch

A Dispatch<T> is an object that allows the client to perform XML level operations. It is the client side equivalent to the Provider<T> interface.

2.2.1. Configure dynamic Service

2.2.2. Create a Dispatch

2.2.3. Invoke the Dispatch

The Dispatch interface provides support for 4 MEP: request-response, asynchronous polling, asynchronous callback

2.2.3.1. Synchronous request response

2.2.3.2. Asynchronous request response

2.2.3.3. One way

2.3. Packaging

Chapter 3. Developing Handlers

3.1. Types

3.1.1. SOAP handlers

3.1.2. Logical handlers

3.2. Defining the handler-chains

3.2.1. Handler chains description file

3.2.1.1. HandlerChain annotation

3.2.2. Defining handlers on the endpoint

3.2.2.1. Using the @HandlerChain annotation

3.2.2.2. Using the webservices.xml

3.2.3. Defining handlers for the consumer

3.2.3.1. Using the @HandlerChain annotation

3.2.3.2. Using the javaee:service-ref element

3.2.3.3. Using the HandlerResolver

Chapter 4. Annotations references

4.1. JWS Annotations (javax.jws)

4.1.1. javax.jws.WebService

4.1.2. javax.jws.WebMethod

4.1.3. javax.jws.WebParam

4.1.4. javax.jws.WebResult

4.1.5. javax.jws.OneWay

4.1.6. javax.jws.HandlerChain

4.1.7. javax.jws.soap.SOAPBinding

4.2. JAX-WS Annotations (javax.xml.ws)

4.2.1. javax.xml.ws.WebServiceProvider

4.2.2. javax.xml.ws.WebServiceRef

4.2.3. javax.xml.ws.WebServiceRefs

4.2.4. javax.xml.ws.WebServiceClient

4.2.5. javax.xml.ws.BindingType

4.2.6. javax.xml.ws.RequestWrapper

4.2.7. javax.xml.ws.ResponseWrapper

4.2.8. javax.xml.ws.RespectBinding

4.2.9. javax.xml.ws.ServiceMode

4.2.10. javax.xml.ws.WebEndpoint

4.2.11. javax.xml.ws.WebFault

4.2.12. javax.xml.ws.Feature

4.2.13. javax.xml.ws.FeatureParameter

4.2.14. javax.xml.ws.Action

4.2.15. javax.xml.ws.FaultAction

4.2.16. javax.xml.ws.soap.Addressing

4.2.17. javax.xml.ws.soap.MTOM

Glossary

JAX-WS Terminology

Endpoint	A Webservice endpoint is the web service implementation. It's a server side artifact. It can be implemented as a POJO or as a Stateless EJB.
Consumer	Also known as a webservice client. It is a client side artifact.
Portable Artifacts	JAX-WS generated artifacts (classes / WSDL / XSD) that are portable across JAX-WS implementation. It means that artifacts generated using Sun RI tools will be usable *without any changes* on any Java EE 5 compliant application server.
Provider	Endpoint can be implemented as Providers if they require low level XML access.
Handler	A Handler can be seen like a webservice interceptor. It can be used on client and/or server side. Two kinds of handlers exists: Logical and SOAP handlers.
Dispatch	A Dispatch is a webservice client working at the XML level. A Dispatch is the client side equivalent to a Provider.
POJO	Plain Old Java Object. Commonly used word meaning that the beans do not have to implement an interface to be usable by the container.
QName	XML Qualified Name. Disambiguate XML names by prefixing the names (local-part) with a namespace (URI).
MEP	Message Exchange Pattern. This is the pattern of the exchange (one-way, request-response, ...).
SEI	Service Endpoint Interface. This is an annotated Java interface that represents the business object.
MTOM	Message Transmission Optimization Mechanism. This is a W3C recommendation defining a method to efficiently sending binary data to and from webservices.
WSDL	Web Service Description Language. This is the cornerstone of the webservices system. It is an XML file that can basically be compared to a Java interface: it's the contract defining the wire exchange (possible operations, expected types and parameters, ...) between a consumer and an endpoint.