

# JAX-WS

## JAVA API FOR XML WEB SERVICES

INTRODUCTION TO JAX-WS, THE JAVA API FOR XML  
BASED WEB SERVICES (SOAP, WSDL)

Peter R. Egli  
INDIGOO.COM

## Contents

1. What is JAX-WS?
2. Glassfish = Reference implementation for JAX-WS
3. Glassfish projects, Java packages
4. POJO / Java bean as web service class
5. JAX-WS JavaBeans versus provider endpoint
6. JAX-WS dispatch client versus dynamic client proxy API
7. JAX-WS development / deployment
8. JAXB – Binding XML documents to Java objects
9. JAXR – JAVA API for XML Registries
10. WSIT – Web Service Interoperability Technologies
11. Short comparison of important Java WS stacks

## 1. What is JAX-WS?

JAX-WS is the **core Java web service technology** (standard for Java EE):

→ JAX-WS is the standard programming model / API for WS on Java (JAX-WS became a standard part of Java as of version 1.6).

→ JAX-WS is **platform independent** (many Java platforms like Glassfish, Axis2 or CXF support JAX-WS). Services developed on one platform can be easily ported to another platform.

→ JAX-WS makes use of **annotations** like **@WebService**. This provides better scalability (no central deployment descriptor for different WS classes is required).

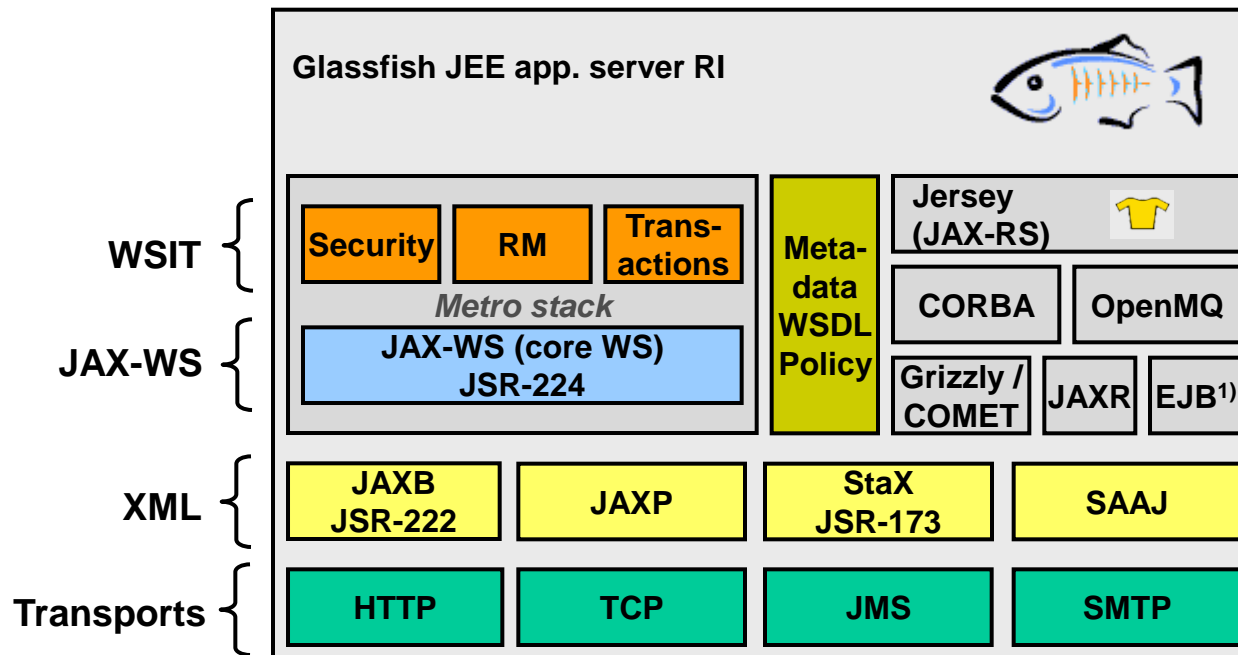
→ JAX-WS uses the **POJO** concept (use of plain Java classes to define web service interfaces).

→ JAX-WS replaces / supersedes JAX-RPC (= old Java web services, basically RMI over web service). JAX-WS is more **document orientated** instead of RPC-oriented.

Glassfish  is the JAX-WS reference implementation (RI, see <https://jax-ws.java.net/>).

## 2. Glassfish = Reference implementation for JAX-WS

- Glassfish:** Java EE reference implementation (RI), reference Java application server.
  - Metro:** WS stack as part of Glassfish (consists of JAX-WS and WSIT components).
  - JAXB:** Data binding (bind objects to XML documents or fragments).
  - SAAJ:** SOAP with Attachments API for Java.
  - JAX-WS:** Java core web service stack.
  - WSIT:** Web Services Interoperability Technologies (enables interop with .Net).
  - JAXR:** Java API for XML Registries (WS registry).
  - StaX:** Streaming API for XML.
- Details see <https://glassfish.java.net/downloads/3.1.1-final.html>



1) Local container only

## 3. Glassfish projects, Java packages (selection)

Glassfish is developed in various independent projects. Together, they build the Glassfish service platform.

Parent project	Project	Description	Java package
Glassfish	GF-CORBA	CORBA	com.sun.corba.se.*
Glassfish	OpenMQ	JMS	javax.jms.*
Glassfish	WADL	IDL for REST services	org.vnet.ws.wadl.* org.vnet.ws.wadl2java.*
Glassfish	JAX-RS, Jersey (=RI)	REST services	javax.ws.rs.*
jax-ws-xml, jwsdp	JAXB	Java API for XML Binding	javax.xml.bind.*
jax-ws-xml, jwsdp	JAX-RPC	Java API for XML RPC Old Java WS (superseded by JAX-WS)	java.xml.rpc.*
Metro (Glassfish)	JAX-WS	Java API for XML Web Services	javax.jws.* javax.xml.ws.*
Metro (Glassfish)	SAAJ	SOAP attachments	javax.xml.soap.*
Metro (Glassfish)	WSIT (previously Tango)	Web Services Interoperability Technologies (WS-Trust etc.)	Various

## 4. POJO / Java bean as web service class (1/2)

### Web services through simple annotations:

The programming model of JAX-WS relies on simple annotations.

This allows to turn existing business classes (POJOs) into web services with very little effort.

Web service endpoints are either explicit or implicit, depending on the definition or absence of a Java interface that defines the web service interface.

### Implicit SEI (Service Endpoint Interface):

Service does not have an explicit service interface, i.e. the class itself is the service interface.

#### @WebService

```
public class HelloWorld  
{
```

#### @WebMethod

```
public String sayHello() {...}
```

```
public HelloWorld() {...}
```

#### @PostConstruct

```
public void init() {...}
```

#### @PreDestroy

```
public void teardown() {...}
```

```
}
```

Through `@WebService` annotation, the class `HelloWorld` becomes an SEI (Service Endpoint Interface) and all public methods are exposed.

Annotation for individual method to be exposed as a web service method.

The SEI implementing class must have a public default ctor.

Called by container before the implementing class is called the first time.

Called by container before the implementing class goes out of operation.

## 4. POJO / Java bean as web service class (2/2)

### Explicit SEI:

The service bean has an associated service endpoint interface (reference to interface).

```
public interface IHello
```

Explicit endpoint interface.

```
{
```

```
    @WebMethod
```

```
    public String sayHello() {...}
```

```
}
```

```
@WebService(endpointInterface=IHello)
```

Reference to explicit service endpoint interface.

```
public class HelloWorld
```

```
{
```

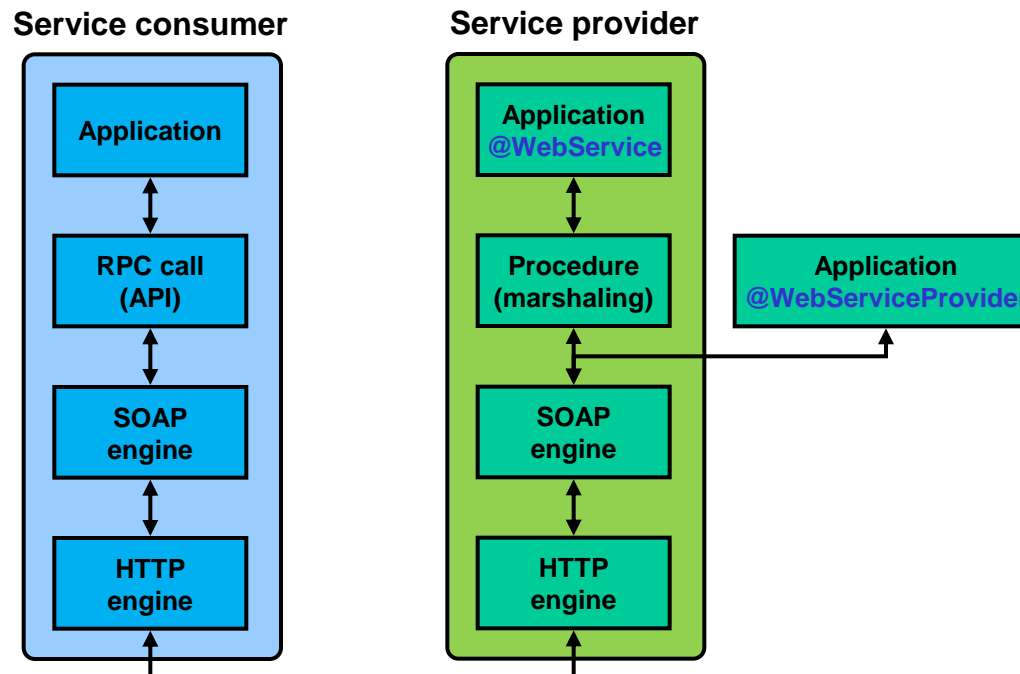
```
    public String sayHello() {...}
```

```
}
```

## 5. JAX-WS JavaBeans versus provider endpoint (1/3)

Web services on the service provider side can be defined at 2 levels:

- The annotation `@WebService` is a high-level annotation to turn a POJO into a web service.
- The `@WebServiceProvider` is lower-level and gives more control over the web service.





## 5. JAX-WS JavaBeans versus provider endpoint (2/3)

### A. JavaBeans endpoints (@WebService annotation):

JavaBeans as endpoints allow to expose Java classes (actually methods) as web services. This hides most of the WS complexity to the programmer.

#### JavaBeans endpoint example:

```
@WebService
public class HelloWorld
{
    @WebMethod
    public String sayHello() {...}
}
```

## 5. JAX-WS JavaBeans versus provider endpoint (3/3)

### B. Provider endpoints (@WebServiceProvider annotation):

Provider endpoints are more generic, message-based service implementations. The service operates directly on (SOAP) message level. The service implementation defines 1 generic method "invoke" that accepts a SOAP message and returns a SOAP result. The provider interface allows to operate a web service in payload mode (service method directly receives XML messages).

### Provider endpoint example:

The web service class has only a single method *invoke*. This method receives the SOAP messages. Decoding of the message takes place in this method.

```
@WebServiceProvider (
    portName="HelloPort",
    serviceName="HelloService",
    targetNamespace="http://helloservice.org/wsdl",
    wsdlLocation="WEB-INF/wsdl/HelloService.wsdl"
)
@BindingType (value="http://schemas.xmlsoap.org/wsdl/soap/http")
@ServiceMode (value=javax.xml.ws.Service.Mode.MESSAGE)
public class HelloProvider implements Provider<SOAPMessage> {

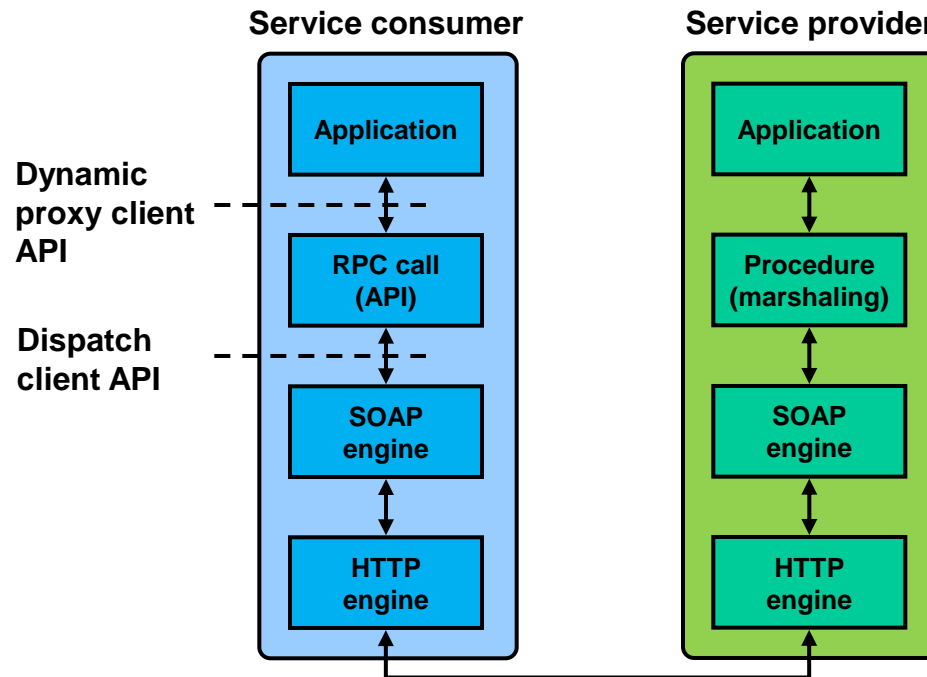
    private static final String helloResponse = ...
    public SOAPMessage invoke(SOAPMessage req) {
        ...
    }
}
```

## 6. JAX-WS dispatch client versus dynamic client proxy API (1/3)

Similar to the server APIs, JAX-WS clients may use 2 different APIs for sending web service requests.

→ A dispatch client gives direct access to XML / SOAP messages.

→ A dynamic proxy client provides an RPC-like interface to the client application.



## 6. JAX-WS dispatch client versus dynamic client proxy API (2/3)

### A. Dispatch client (dynamic client programming model):

The dispatch client has direct access to XML (SOAP) messages. The dispatch client is responsible for creating SOAP messages. It is called dynamic because it constructs SOAP messages at runtime.

### Dispatch client example (source: axis.apache.org):

```
String endpointUrl = ...;
QName serviceName = new QName("http://org/apache/ws/axis2/sample/echo/", "EchoService");
QName portName = new QName("http://org/apache/ws/axis2/sample/echo/", "EchoServicePort");
/** Create a service and add at least one port to it. */
Service service = Service.create(serviceName);
service.addPort(portName, SOAPBinding.SOAP11HTTP_BINDING, endpointUrl);
/** Create a Dispatch instance from a service.*/
Dispatch<SOAPMessage> dispatch = service.createDispatch(portName, SOAPMessage.class, Service.Mode.MESSAGE);
/** Create SOAPMessage request message. */
MessageFactory mf = MessageFactory.newInstance(SOAPConstants.SOAP_1_1_PROTOCOL);
// Create a message. This example works with the SOAPPART.
SOAPMessage request = mf.createMessage();
SOAPPart part = request.getSOAPPart();
// Obtain the SOAPEnvelope and header and body elements.
SOAPEnvelope env = part.getEnvelope();
SOAPHeader header = env.getHeader();
SOAPBody body = env.getBody();
// Construct the message payload.
SOAPElement operation = body.addChildElement("invoke", "ns1", "http://org/apache/ws/axis2/sample/echo/");
SOAPElement value = operation.addChildElement("arg0");
value.addTextNode("ping");
request.saveChanges();
/** Invoke the service endpoint. */
SOAPMessage response = dispatch.invoke(request);
```

## 6. JAX-WS dispatch client versus dynamic client proxy API (3/3)

### B. Dynamic proxy client (static client programming model):

A dynamic proxy client is similar to a stub client in the JAX-RPC programming model (=RPC model). A dynamic proxy client uses a service endpoint interface (SEI) which must be provided. The dynamic proxy client classes are generated at runtime (by the Java dynamic proxy functionality).

### Dynamic proxy client example:

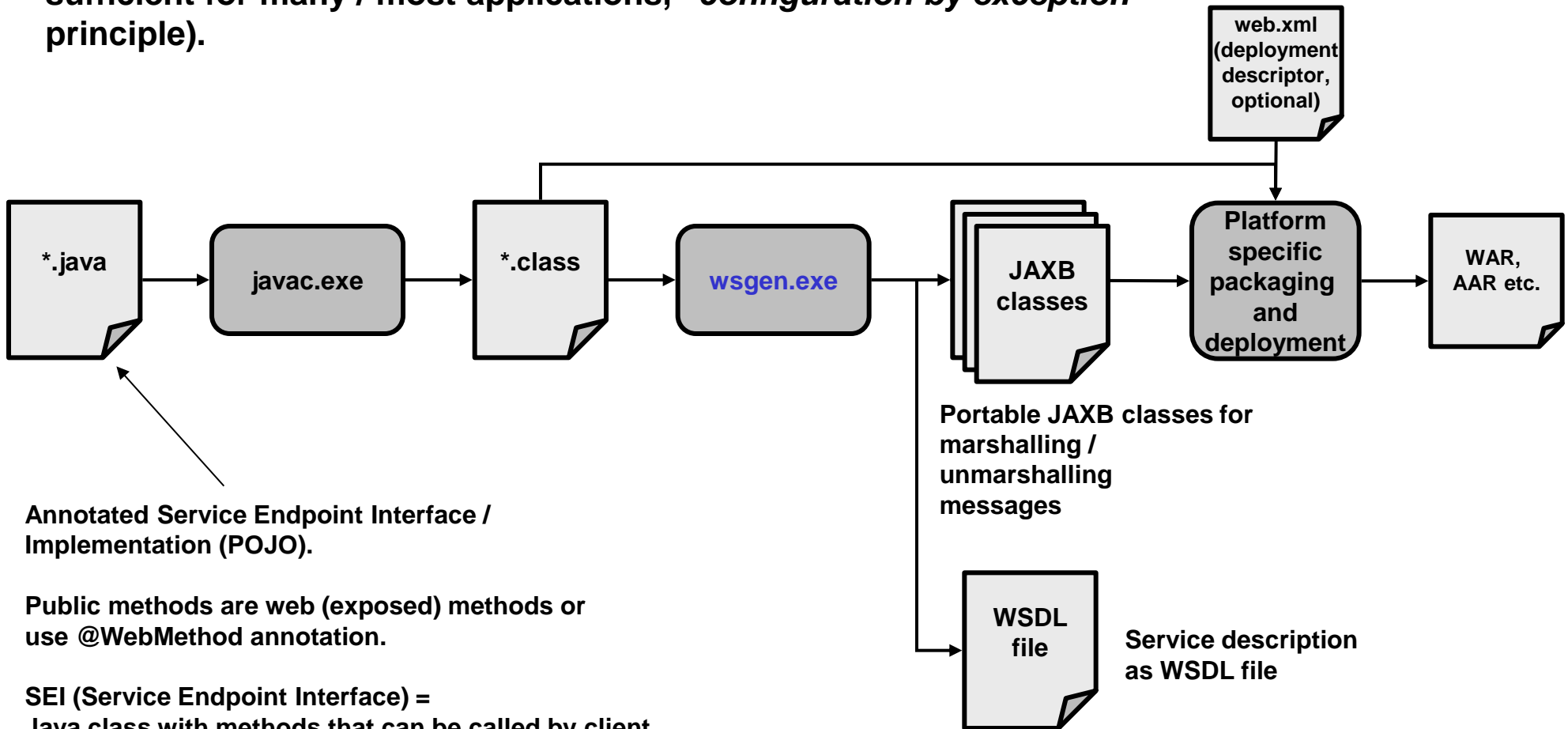
```
EchoService port = new EchoService().getEchoServicePort();  
String echo = port.echo("Hello World");
```

## 7. JAX-WS development / deployment (1/2)

### 1. Start with POJO / bean class as service class (bottom-up development):

Server stub files and (optionally) WSDL files are created from Java class file(s).

The deployment descriptor web.xml is optional (JAX-WS defaults are sufficient for many / most applications, "*configuration by exception*" principle).



Annotated Service Endpoint Interface / Implementation (POJO).

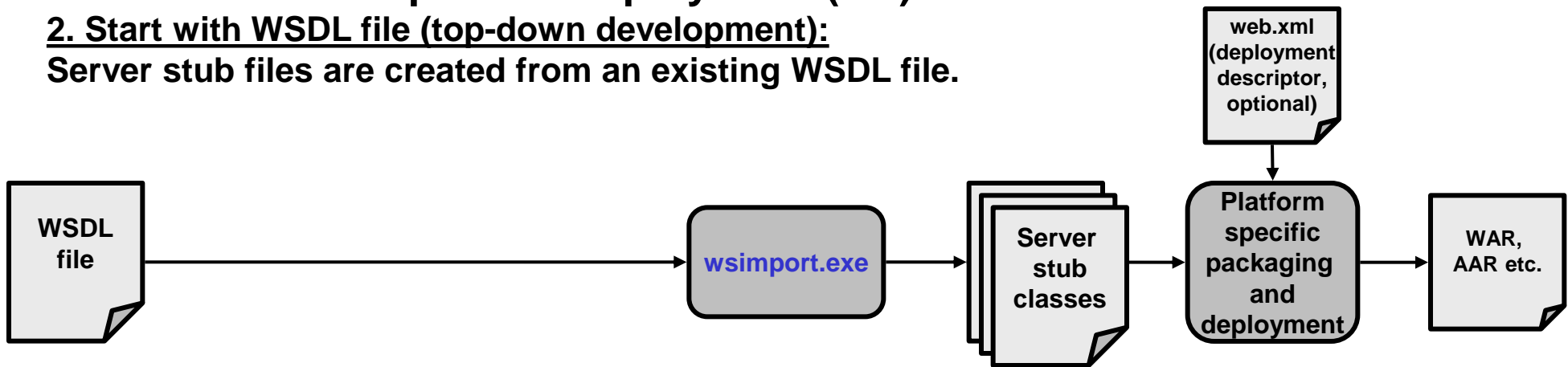
Public methods are web (exposed) methods or use @WebMethod annotation.

SEI (Service Endpoint Interface) = Java class with methods that can be called by client.

## 7. JAX-WS development / deployment (2/2)

### 2. Start with WSDL file (top-down development):

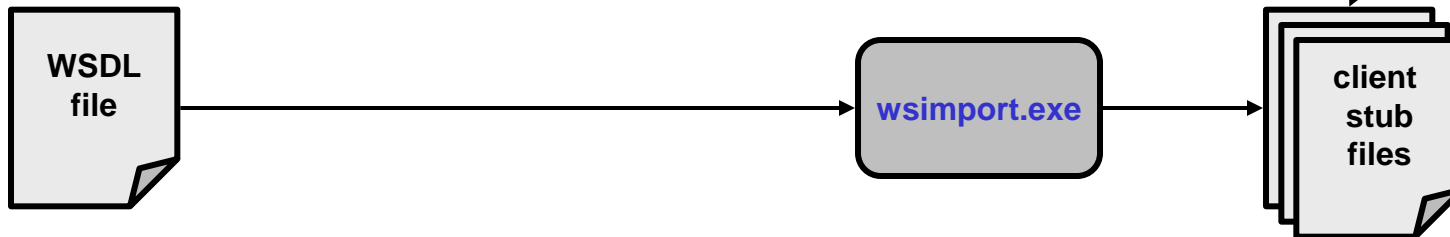
Server stub files are created from an existing WSDL file.



Portable artefacts for client & server:

- Service class
- SEI class
- Exception class
- JAXB classes for marshalling messages

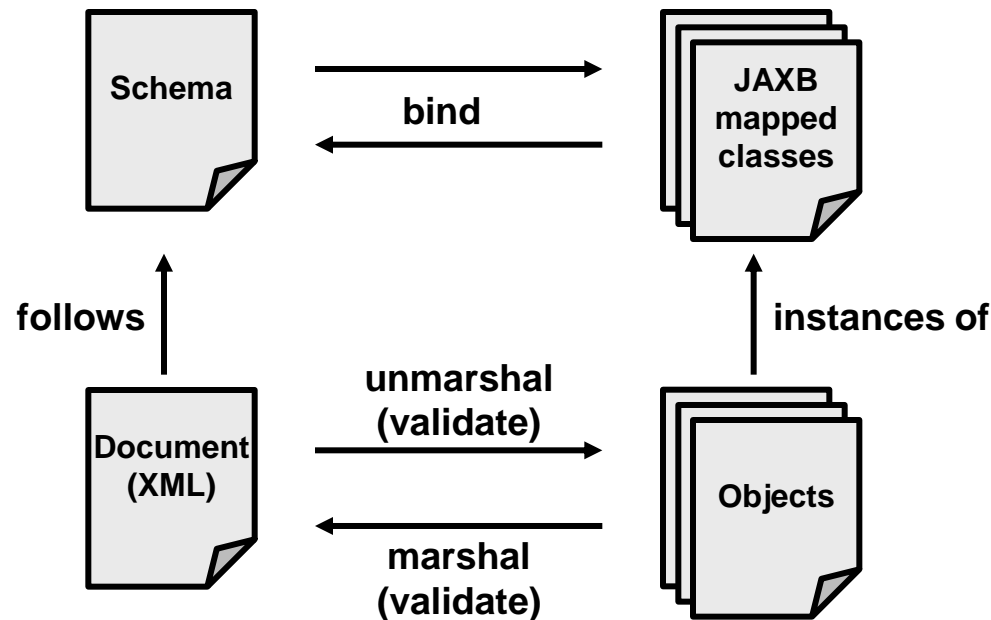
### 3. Creation of client stub files from a WSDL file (dynamic proxy client):



## 8. JAXB – Binding XML documents to Java objects

JAX-WS uses JAXB for the binding between Java objects and XML documents. The binding / JAXB details are hidden from the JAX-WS programmer.

Relationship between schema, document, Java objects and classes with JAXB:



JAXB mapping of XML schema built-in types to Java built-in types (selection):

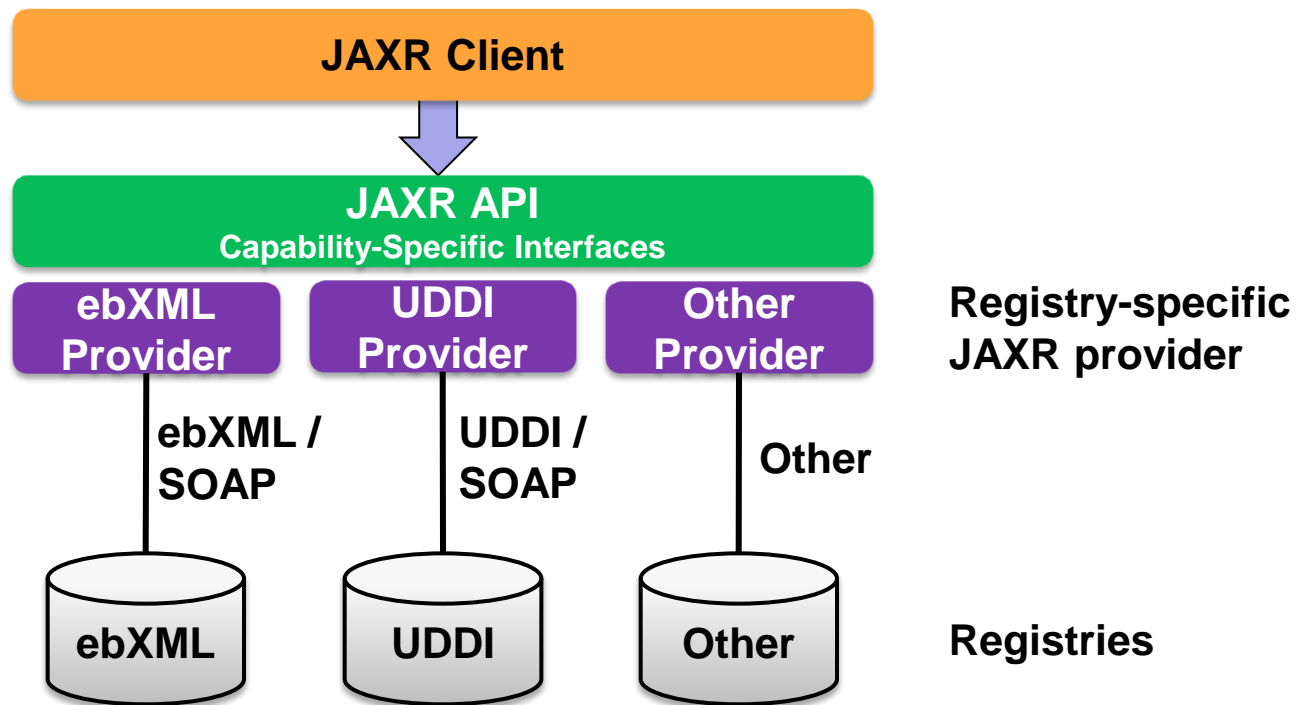
XML schema type	Java data type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short



## 9. JAXR – JAVA API for XML Registries

JAXR is a uniform and standard API for accessing different kinds of XML registries. XML registries are used for registering (by server / service) and discovering (by client) web services.

JAXR follows the well-known pattern in Java with a service provider interface (SPI for registering XML metadata) and a client interface (see also JMS or JNDI).



Source: docs.oracle.com

## 10. WSIT – Web Service Interoperability Technologies (1/2)

Interoperability is crucial for web services (web services in general claim platform independence and interoperability).

WSIT is the standard Java WS protocol stack beyond basic WS functionality (SOAP, WSDL) to achieve interoperability between Java and .Net (for more complex applications that go beyond simple WS requests).

JAX-WS defines the WS core stack while WSIT provides many of the WS-\* standards required for interoperability.

WSIT is an open source project started by Sun Microsystems to promote WS-interoperability with .Net 3.0 (WCF).

WSIT is bundled inside the Metro WS stack. The Metro stack in turn is bundled inside Glassfish (see Slide 2).

WS-I (Web Service Interoperability) defines profiles (sets of WS standards, see [www.oasis-ws-i.org/](http://www.oasis-ws-i.org/)) for which WS stacks can claim compatibility such as:

WS-I Basic Profile 1.1:

SOAP 1.1 + HTTP 1.1 + WSDL 1.1

WS-I Basic Profile 1.2:

SOAP 1.1 + HTTP 1.1 + WSDL 1.1 + WS-Addressing 1.0 + MTOM 1.0

WS-I Simple SOAP Binding Profile 1.0:

SOAP 1.1 + HTTP 1.1 + WSDL 1.1 + UDDI

**WSIT versus WS-I?**

**WS-I defines profiles while WSIT implements them.**

## 10. WSIT – Web Service Interoperability Technologies (2/2)

Overview of WS-standards compliance by JAX-WS / Metro-stack and WCF:

Standard	JAX-WS (Metro@Glassfish)	Microsoft WCF	Description
WS-I Basic Profile	Yes (JAX-WS)	Yes	SOAP 1.1 + HTTP 1.1 + WSDL 1.1.
WS-MetadataExchange	Yes (WSIT)	Yes	Exchange of web service meta-data files.
WS-Transfer	Yes (WSIT)	Yes	Protocol for accessing XML-representations of WS resources.
WS-Policy	Yes (WSIT)	Yes	XML-representations of WS policies (security, QoS).
WS-Security	Yes (WSIT)	Yes	Framework protocol for applying security to web services.
WS-SecureConversation	Yes (WSIT)	Yes	Protocol for sharing security contexts between sites.
WS-Trust	Yes (WSIT)	Yes	Protocol for establishing trust between WS provider and consumer.
WS-SecurityPolicy	Yes (WSIT)	Yes	Set of security policy assertions to be used with WS-Policy.
WS-ReliableMessaging	Yes (WSIT)	Yes	Protocol for reliable WS message delivery (in-order, exactly-once).
WS-RMPolicy	Yes (WSIT)	Yes	XML-representations of WS policies of RM-WS-endpoints.
WS-Coordination	Yes (WSIT)	Yes	Protocol for coordinating web service participants (providers, consumers).
WS-AtomicTransaction	Yes (WSIT)	Yes	Protocol for atomic transactions with groups of web services.
WS-BusinessActivity	No	Yes (.Net 4.0)	Business activity coordination to be used in the WS-Coordination framework.
WS-Eventing	No	No	Protocol for subscribe / publish to WS for receiving notifications.
WS-Notification	No	No	Protocol for subscribe / publish to topic-based WS for receiving notifications.

## 11. Short comparison of important Java WS stacks

### 1. JAX-WS (Glassfish RI):

- JAX-WS compliant (the reference implementation defines the standard).
- Supported by many WS frameworks.

### 2. Apache CXF framework:

- Easy to use, slim.

### 3. Apache Axis2:

- Features-rich, but also high complexity.
- Not (fully) JAX-WS compliant (not JAX-WS technology compatibility kit compliant).
- Support for different XML-Java binding frameworks (XMLBeans etc.).

### 4. JBoss:

- Actually uses Glassfish / Metro as WS-stack (<http://jbossws.jboss.org/>).

### 5. Spring WS:

- Contract-first development (first WSDL, then code).
- Support for different XML-Java binding frameworks (XMLBeans, JiXB etc.).

Comparison of Java WS stacks see also <http://wiki.apache.org/ws/StackComparison>.