

The Web Services Programming Model

JAX-RPC and JSR 109

Denise Hatzidakis

Perficient, Inc.

denise.hatzidakis@perficient.com





The Web Service Solution

- There are three major parts to a Web Services Solution
 - A Programming model
 - How you write clients to access Web services
 - How you write service implementations
 - How you handle other parts of the SOAP spec (headers, attachments, *etc.*)
 - A deployment model
 - Deployment descriptors to map service implementations to SOAP messages
 - Type mapping for parameters (at least in RPC)
 - An Engine
 - Code to receive SOAP messages and invoke service implementations
 - Code to map Java types to XML

Java Standard Programming and Deployment Model

- Portable Web Services applications (JSR 101) – JAX/RPC
 - Defines the mapping of WSDL to Java and vice versa
 - Defines a client API to invoke a remote Web service
 - Defines a runtime environment for Java Bean as an implementation of a Web service
 - Handler model

- Standard Web Services Deployment Descriptors (JSR 109)
 - JSR109 conceptually enhances JSR101
 - Defines a J2EE compliant deployment/packaging model for Web Services on the server side and for the client
 - New deployment descriptors for Web services
 - Server-side programming model
 - Stateless Session EJB as implementation of a Web service
 - Client-side programming model
 - EJB, Servlet/JSPs, Application Client as client to Web Services
 - J2EE Container required

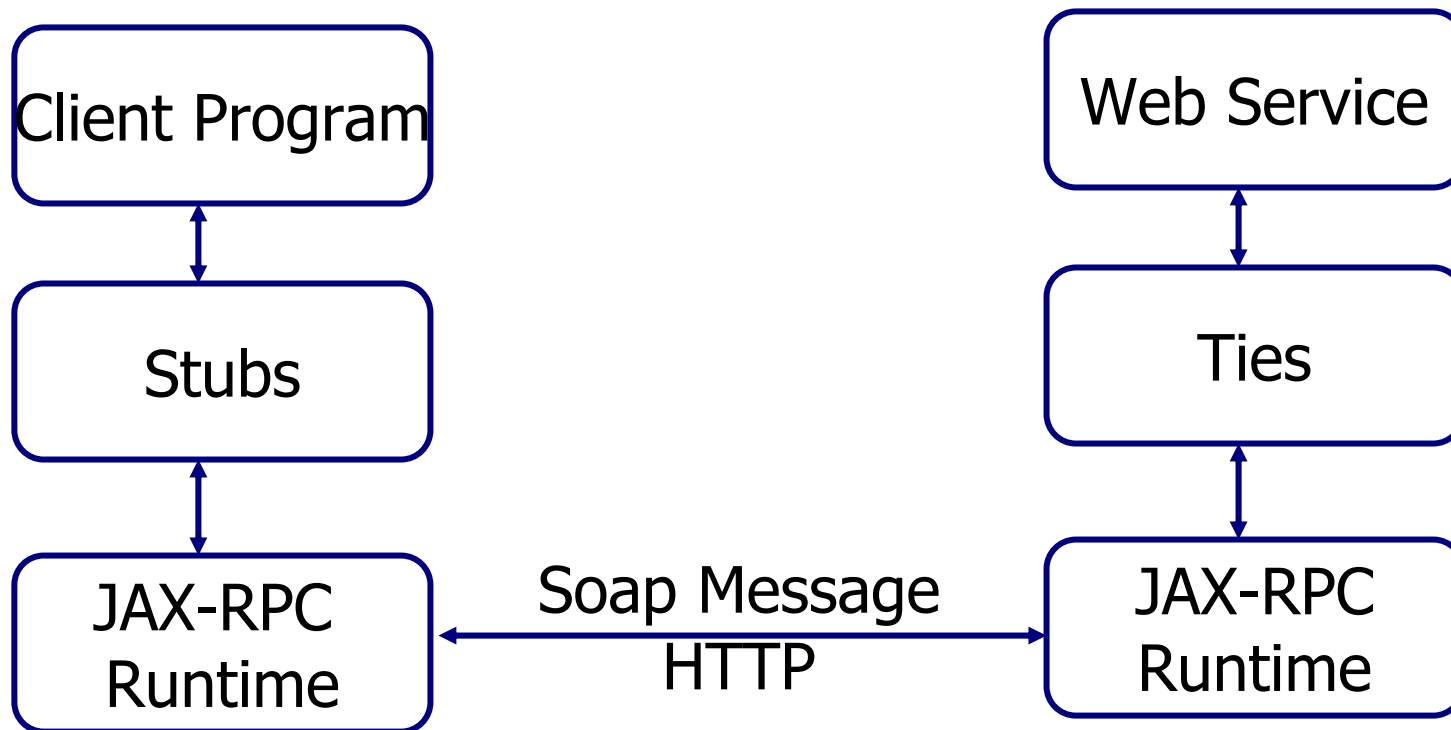
JSR 101 – JAX RPC – Standard Programming Model

- JSR-101
- Version 1.0
 - Version 1.1 will be coming out shortly and will be part of J2EE1.4
 - Version 1.1 provides WS-I compliance support
- Has a strong RMI flavor
- Oriented around WSDL
- Requires support for rpc/encoded, document/literal
- JAX-RPC is designed for portability of code across multiple vendors
 - Any client or service implementation should be portable across any vendor that supports JAX-RPC

JSR 101 – JAX RPC – Standard Programming Model

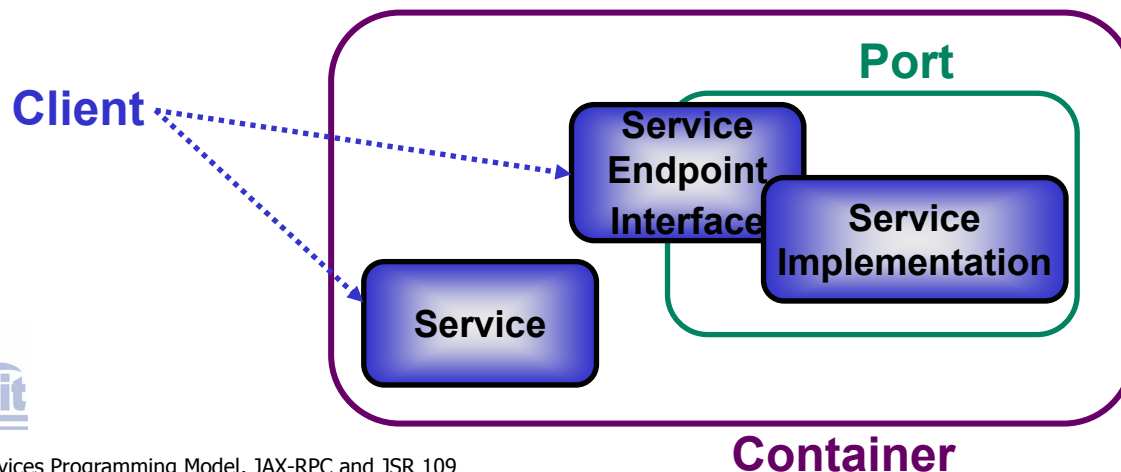
- JAX-RPC is meant to be protocol-neutral
 - but only defines support for SOAP over HTTP
- Features
 - Defines Standard Java APIs for XML based RPC (Remote Procedure Call)
 - Defines a standardized mapping model from Java artifacts
 - Client APIs to access a Web service for non-container-based clients
 - Handler model

JAX-RPC Working Model



An Architectural View

- JAX-RPC and JSR-109 fit together to form a standard approach for Web service implementation, access and deployment in Java
- Based on a few concepts
 - A **Service Implementation** (either a Java Bean or Stateless Session Bean) implements the methods of a WSDL-Described interface
 - A **Service Endpoint Interface (SEI)** provides a Java "view" of the WSDL-described interface. Clients interact with objects implementing this interface
 - A **Service** mediates access to the **Ports** (acts as a factory for ports)





JAX-RPC

- Mappings for WSDL to XML
 - Standard Java Type <-> XML type mapping
 - Standard Java Type <-> WSDL type mapping
 - Standard SOAP binding mapping
- SOAP Manipulation Library
 - JAX-RPC is built on SAAJ (SOAP Attachments API for Java)
 - SAAJ was formerly part of JAXM
 - Provides pattern for sending and receiving SOAP messages with or without attachments, synchronous or asynchronous
- Servlet Container Service Endpoint
 - Standard way to specify servlets that receive SOAP messages
 - Compatible with JSR 109 style deployment descriptors
- Basic Authentication



Analogies to EJBs

- The JAX-RPC approach is analogous to the EJB (or RMI) approach in a number of ways
- A **Service Endpoint Interface** (SEI) is analogous to the Remote interface of an EJB
 - Both the Stub and the Service Implementation implement the methods of the SEI
 - If the service implementation is an EJ, it does not directly implement the SEI interface, just like the EJB does not directly implement the remote interface
 - The Service Implementation is thus analogous to the Bean Implementation of an EJB
- ServiceFactories (and Services) are analogous to the EJB Homes
 - You obtain stubs from them – they are factories of remote objects



WSDL Document Overview

- **Definition**
 - The root of the WSDL document
 - Contains the definition of one or more services
 - Usually contains attributes
- **Service**
 - Defines the service
- **Messages and PortTypes**
 - Describes the actions available for the service
- **Bindings**
 - How to communicate with the service

WSDL Review

A WSDL document defines Web Service *via*:

Messages

Defines a single interaction with the service

Types

Defines data types used in a message

Operations

Description of an action

Port Types

Describes the set of operations supported by the service.

Bindings

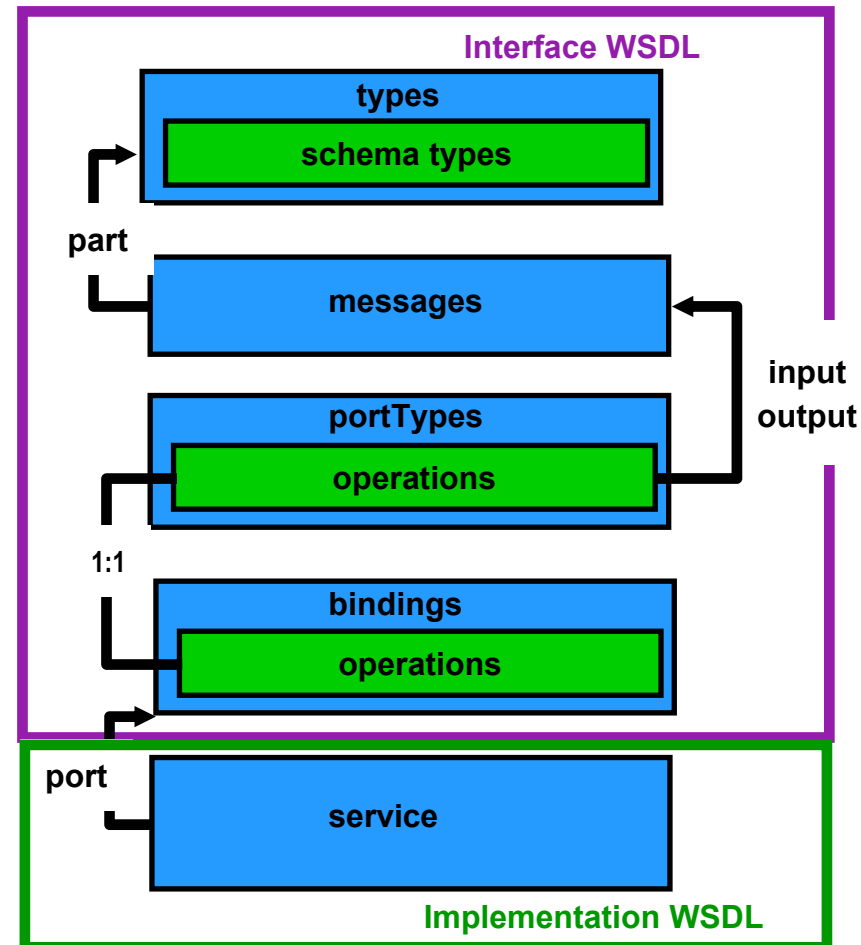
A concrete protocol and data format for a particular port type.

Port

Describes the network address where the service is being hosted.

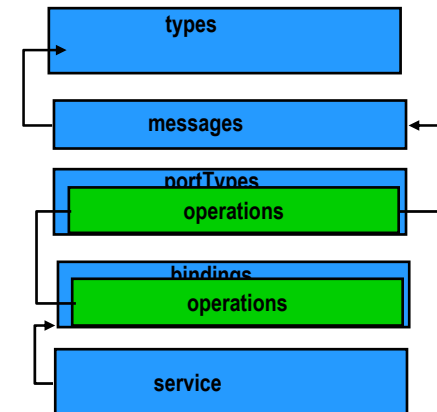
Service

Ties together all the elements of the service.



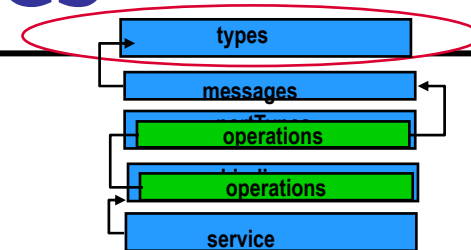
<definitions> element attributes

- **targetNamespace**
 - Chosen to be unique for this service
 - **targetNamespace**="http://employee.webservices.wsad.ibm.com"
- **xmlns**
 - Default namespace – the WSDL defined name space
 - **xmlns**="http://schemas.xmlsoap.org/wsdl/"
- **xmlns:apachesoap**
 - Apache SOAP Type namespace
 - **xmlns:apachesoap**="http://xml.apache.org/xml-soap"
- **xmlns:impl**
 - Implementation namespace
 - **xmlns:impl**="http://employee.webservices.wsad.ibm.com"
- **xmlns:intf**
 - Interface namespace
 - **xmlns:intf**="http://employee.webservices.wsad.ibm.com"
- **xmlns:wSDL**
 - WSDL namespace
 - **xmlns:wSDL**="http://schemas.xmlsoap.org/wsdl/"
- **xmlns:wSDLsoap**
 - SOAP namespace
 - **xmlns:wSDLsoap**="http://schemas.xmlsoap.org/wsdl/soap/"
- **xmlns:xsd**
 - Schema namespace
 - **xmlns:xsd**="http://www.w3.org/2001/XMLSchema"



<types> element attributes

- Contains data type definitions other than those defined by the base schema
- May be imported into the WSDL document



```

<wsdl:types>
  <schema elementFormDefault="qualified"
    targetNamespace="http://employee.webservices.wsad.ibm.com"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <complexType name="Employee">
      <sequence>
        <element name="department" nillable="true" type="string"/>
        <element name="phoneNumber" nillable="true" type="string"/>
        <element name="lastName" nillable="true" type="string"/>
        <element name="hireDate" nillable="true" type="dateTime"/>
        <element name="firstName" nillable="true" type="string"/>
        <element name="salary" type="double"/>
        <element name="employeeNo" nillable="true" type="string"/>
      </sequence>
    </complexType>
    <element name="getEmployeeResponse">
      <complexType>
        <sequence>
          <element name="getEmployeeReturn" nillable="true" type="intf:Employee"/>
        </sequence>
      </complexType>
    </schema>
  </wsdl:types>

```

<message> and <part> Elements

- **<message>**

A single piece of information moving between the requester and provider

A single interaction between requester and provider

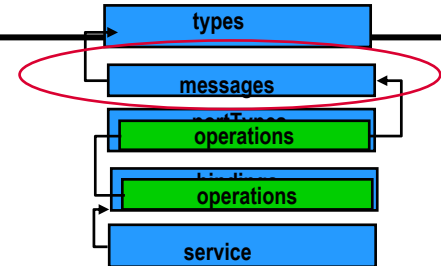
By convention

methodName**Request** and methodName**Response**

- **<part>**

Describes a piece of data associated with the message

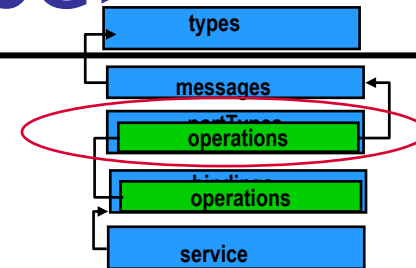
Optional



```
<message name="getEmployeeRequest">
  <part name="lastName" element="impl:getEmployee"></part>
</message>
<message name="getEmployeeResponse">
  <part name="result" element="impl:getEmployeeResponse"></part>
</message>
```

<operation> and <portType>

- Defines what operations the Web Service provides
- <operation>
 - An action
 - Like a Java Method
 - Three messages
 - input message
 - output message
 - fault message
- <portType>
 - A collection of operations
 - Like a Java Class

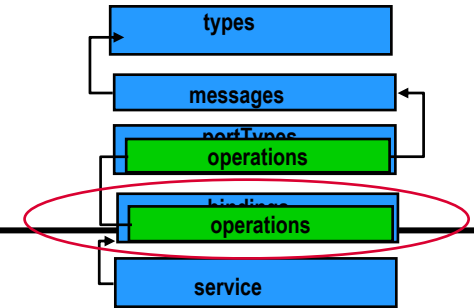


```

<message name="getEmployeeRequest">
  <part name="lastName"
        element="impl:getEmployee"></part>
</message>
<message name="getEmployeeResponse">
  <part name="result"
        element="impl:getEmployeeResponse"></part>
</message>

<portType name="EmployeeManager">
  <operation name="getEmployee">
    <input message="impl:getEmployeeRequest"
           name="getEmployeeRequest"> </input>
    <output message="impl:getEmployeeResponse"
            name="getEmployeeResponse"></output>
  </operation>
</portType>
  
```

<binding> Elements



■ <binding>

- **HOW** the operation is invoked
- Ties to the <portType> element to the protocol defined for the binding
- Defines the transport **protocol** and **style** (rpc or document)
- If a service supports more than one protocol then there should be multiple bindings for the port type.
- For each operation of the <portType> each <message> is detailed
- For SOAP, the <soap:body> element defines **use** (encoding or literal) and **namespace**
- If using encoding, <soap:body> specifies **encodingStyle**

```

<portType name="EmployeeManager">
  <operation name="getEmployee">
    <input message="impl:getEmployeeRequest"
      name="getEmployeeRequest"> </input>
    <output message="impl:getEmployeeResponse"
      name="getEmployeeResponse"></output>
  </operation>
</portType>

<binding name="EmployeeManagerSOAPBinding"
  type="impl:EmployeeManager">
  <wsdlsoap:binding
    transport=http://schemas.xmlsoap.org/soap/http
    style="document"/>
  <wsdl:operation name="getEmployee"><wsdlsoap:operation />
    <input name="getEmployeeRequest">
      <wsdlsoap:body use="literal"/>
    </input>
    <output name="getEmployeeResponse">
      <wsdlsoap:body use="literal"/>
    </output>
  </wsdl:operation>
</binding>

```

<service> Element attributes

- **<service>**

WHERE the service is located

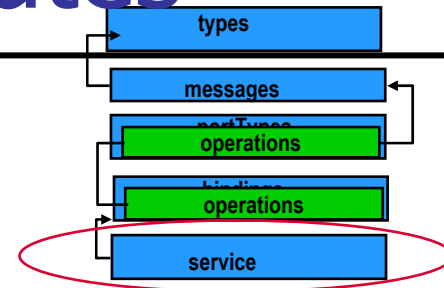
A collection of <ports>s

- **<port>**

defines the availability of a particular binding at a specific endpoint

binding attribute must correspond to a <binding> element

<soap:address> defines the actual location of the service



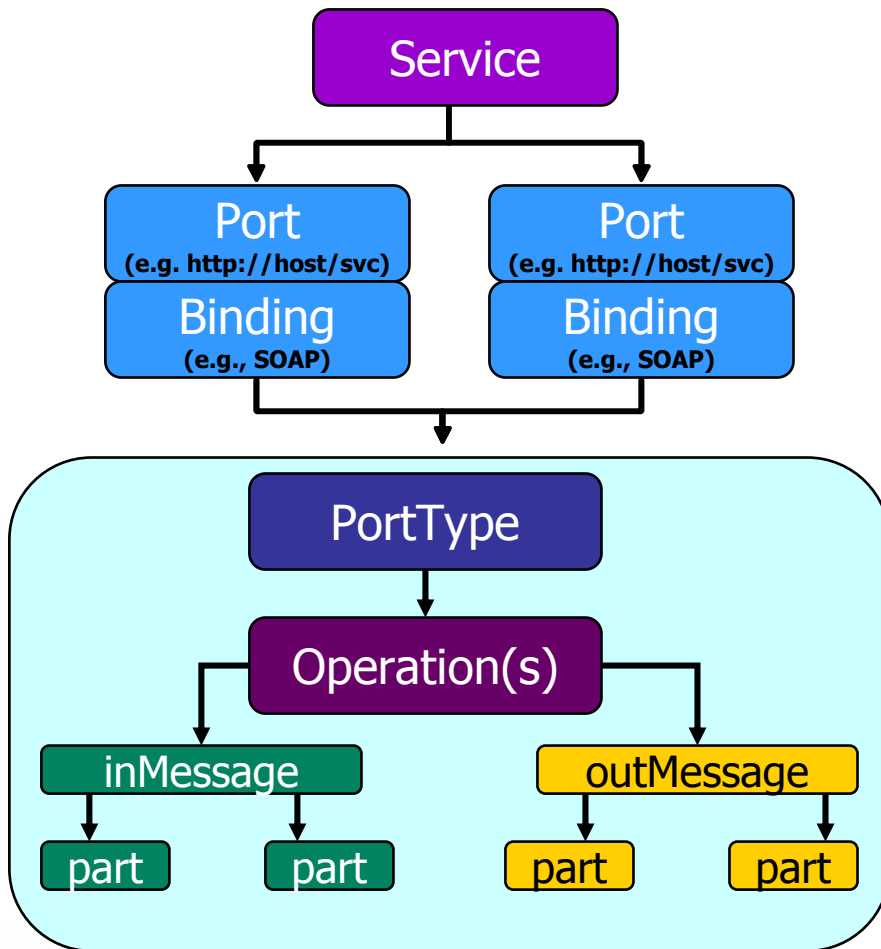
```
<binding name="EmployeeManagerSOAPBinding" type="impl:EmployeeManager">
</binding>

<service name="EmployeeManagerService">
  <port name="EmployeeManagerServicePort" binding="impl:EmployeeManagerSOAPBinding">
    <wsdlsoap:address
      location="http://localhost:9080/WSWSDLWeb/services/EmployeeManager" />
  </port>
</service>
```

WSDL <-> Java Mapping (JAX-RPC)

- Each WSDL <portType> element is mapped to a Java interface.
 - Called the “Service Endpoint Interface” or SEI.
 - This interface must extend `java.rmi.Remote`.
 - Its package name can be derived from the namespace of the <portType> element and vice versa.
- Each <operation> within the <portType> is mapped to a Java method in the service endpoint interface.
 - Every method must throw `java.rmi.RemoteException`.
- The <message> elements of the <operation> are mapped to parameters and return types of the methods of the service endpoint interface.
- The types in the parts map to Java types
 - Complex types must implement `Serializable`
 - JAX-RPC does not define a pass-by-value semantic thus requiring parameters to be serializable

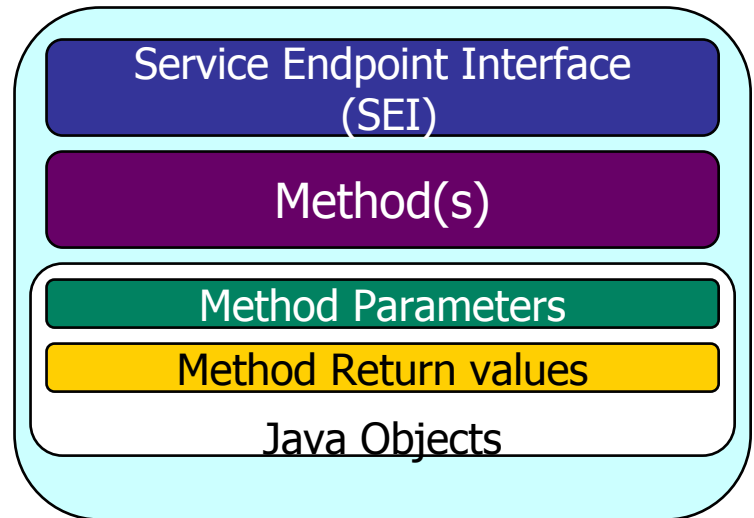
WSDL and JAX/RPC



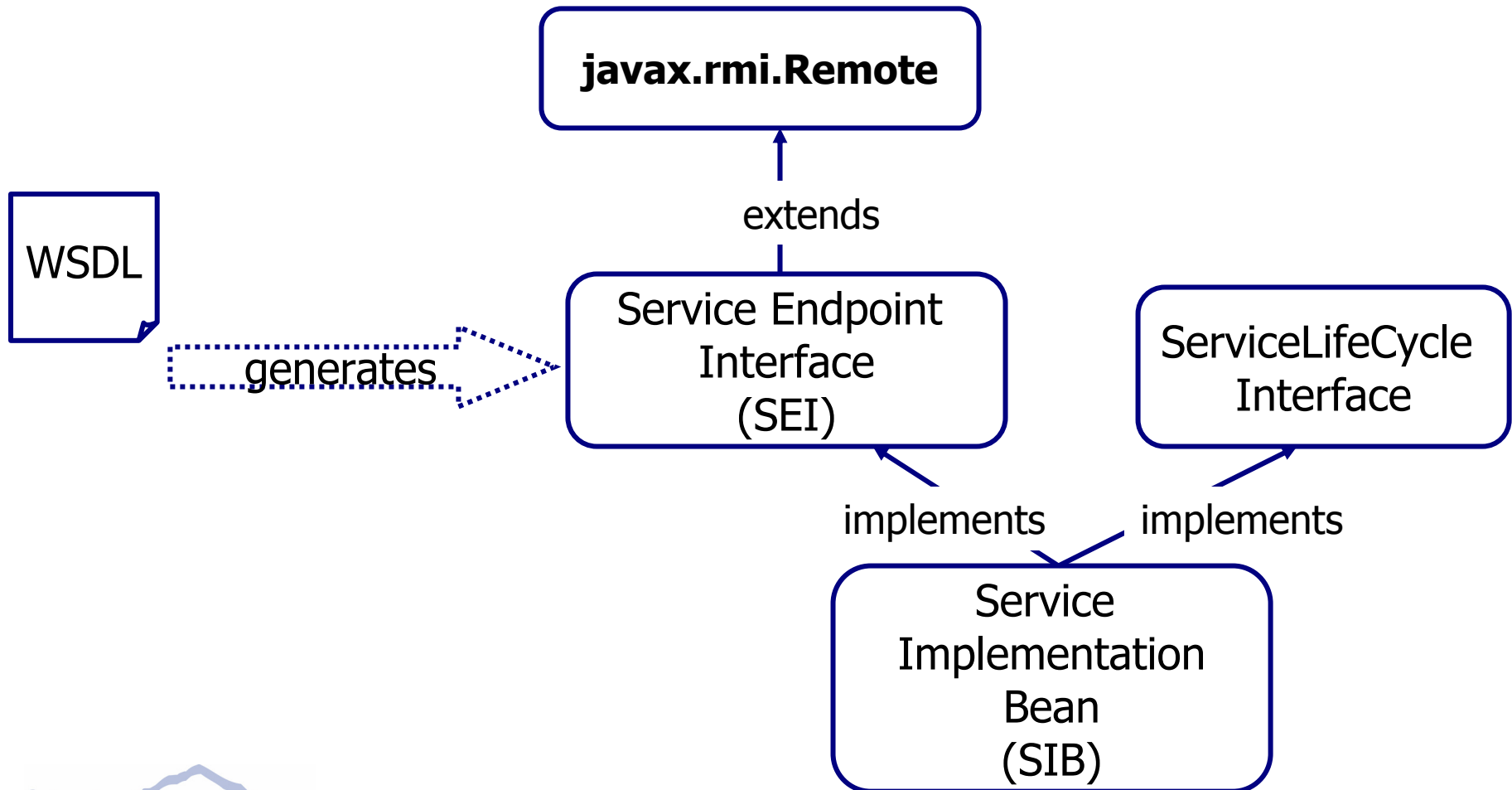
Abstract definition of the service

Service Implementation Bean (SIB)

Client API for invoking services



Service Implementation Interface Hierarchy

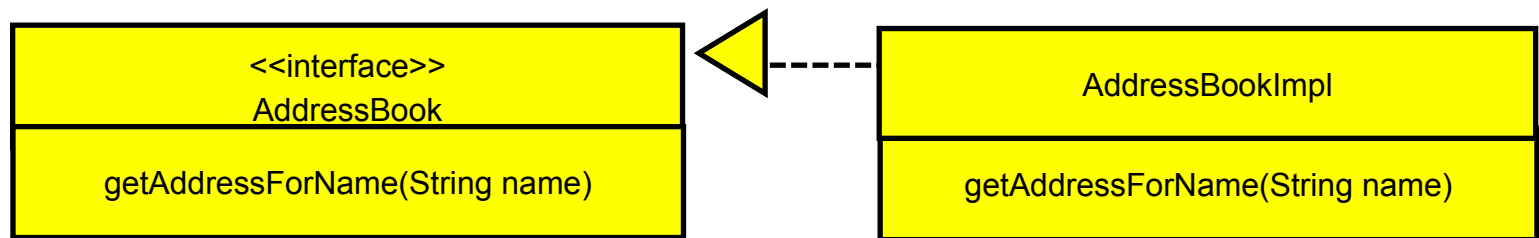


JAX-RPC Server Programming Model

- Port components
 - Service Endpoint Interface (SEI)
 - Service Implementation Bean (SIB)
- Service Implementation Bean
 - Implements the SEI
 - Must be stateless
 - May implement the ServiceLifecycle interface for managing the service's life cycle
- EJB tier (JSR109 defines)
 - SIB must also be a Stateless Session Bean
 - Implementing of the SEI *is not required*
 - Discouraged because SEI methods throw RemoteException
- Web Tier
 - Java Bean implementation

Implementing a JAX/RPC service

- A Quick example will probably best show how the pieces fit.
- Imagine that we want an AddressBook Web Service
 - Can get Addresses for a person's name
- Begin by creating an implementation (either a Stateless Session Bean or a Java Bean) and then creating an SEI for the exposed methods
 - In our example we will call the SEI AddressBook and the Implementation AddressBookImpl
 - The bean implementation must implement (through the implements keyword) the SEI but EJBs don't have to actually implement the SEI, but it can (but the method signatures must match – much like with EJB's)





JAX/RPC Server Example

```
public interface AddressBook extends java.rmi.Remote {  
    public Address getAddressForName(String name);  
}
```

Service Endpoint
Interface (SEI)

```
public class AddressBookImpl implements AddressBook {  
    public Address getAddressForName(String name)  
        throws RemoteException {  
        // fill in implementation here  
    }  
}
```

Service Implementation
Bean (SIB)

```
public class Address implements java.io.Serializable {  
    int name;  
    String streetAddress;  
    ...  
    // getters and setters not shown  
}
```

Java classes which are
mapped to XML must
implement Serializable

Type Mapping

-Primitive Types

XML Schema Type	Java Type
xsd:string	java.lang.String
xsd:integer	java.math.BigInteger
xsd:int	int
xsd:long	long
xsd:short	short
xsd:decimal	java.math.BigDecimal
xsd:float	float
xsd:double	double
xsd:byte	byte
xsd:QName	javax.xml.namespace.QName
xsd:dateTime	java.util.Calendar
xsd:base64Binary	byte[]
xsd:hexBinary	byte[]

- JAX-RPC does not dictate a specific Java mapping for `xsd:anyType`.
- XML Schema simple types which map to Java base types map to the Java wrapper classes instead when an element is marked as nillable in the XML Schema.

Example:

```
<xsd:element name="code"
type="xsd:int" nillable="true" />
```

Maps to: ***java.lang.Integer***



SOAP Encoded Type Mapping

SOAP Encoded Simple Type	Java Type
soapenc:string	java.lang.String
soapenc:boolean	java.lang.Boolean
soapenc:float	java.lang.Float
soapenc:double	java.lang.Double
soapenc:decimal	java.math.BigDecimal
soapenc:int	java.lang.Integer
soapenc:short	java.lang.Short
soapenc:byte	java.lang.Byte
soapenc:base64	byte[]



Note: the SOAP encoded types are all nillable, so they are mapped to the Java wrapper classes



Complex Type Mapping

- XML Arrays
 - XML Arrays are mapped to Java arrays
 - Types of XML Arrays that are mapped
 - Arrays declared using `wsdl:arrayType` or `soapenc:Array`
 - Not recommended by the WS-I Basic Profile
- XML Enumeration
 - Mapped to Java class following enumeration pattern



Complex Type Mapping

- XML Struct / Complex Type
 - xsd:complexType with sequence of elements with simple/complex type
 - xsd:complexType with xsd:all of elements of simple/complex type
 - Mapped to Java classes (called JAX-RPC Value types)
 - with getters and setters to access each element in the complex type
 - must have public default constructor
 - must implement java.io.Serializable
 - not required to be a JavaBean
 - may not implement java.rmi.Remote
 - Notes:
 - no sequencing maintained in Java class
 - if maxOccurs on element > 1, maps to an Array for the element
 - XML element attributes not specified, maps to SOAPElement

Complex Type Mapping

```
<xsd:complexType name = "Book">
  <sequence>
    <element name = "author"
              type = "xsd:string"
              maxOccurs = "10" />
    <element name = "price"
              type = "xsd:float" />
  </sequence>
</xsd:complexType>
```

XMLSchema Fragment

```
Public class Book {
  private float price;
  private String[] author;

  String[] getAuthor() {.....}
  public setAuthor(String[] author)
  {.....}

  public float getPrice() {.....}
  public void setPrice(float price)
  {.....}
}
```

Java

Enumeration Mapping

```
<simpleType
  name="EyeColorType">
  <restriction base="xsd:string">
    <enumeration
      value="green"/>
    <enumeration
      value="blue"/>
  </restriction>
</simpleType>
```

XMLSchema Fragment

```
public class EyeColorType {
  // Constructor
  protected EyeColorType(String value) { ... }

  public static final String _green = "green";
  public static final String _blue = "blue";
  public static final EyeColorType green =
    new EyeColorType(_green);
  public static final EyeColorType blue =
    new EyeColorType(_blue);

  public String getValue() { ... }
  public static EyeColorType fromValue(String value)
    { ... }

  public boolean equals(Object obj) { ... }
  public int hashCode() { ... }
  // ... Other methods not shown
}
```



JAX-RPC and Literal Encoding

- If JAX-RPC specifies a mapping for the XML type of a message part
 - That mapping is used
- If there is no JAX-RPC mapping for the XML type of the message part
 - An implementer of `javax.xml.soap.SOAPElement` is used
 - `SOAPElement` interface comes from SAAJ and provides a DOM-like API for manipulating SOAP messages
- This is true for document or rpc style



JAX/RPC Parameter Modes

- *IN type:*
 - An *IN* parameter is passed as a copy. The value of the *IN* parameter is copied before a Web service invocation. The return value is created as a copy and returned to the Web service client.
- *OUT type:*
 - An *OUT* parameter is passed as a copy without any input value to the Web service method. The Web service method fills out the *OUT* parameter and then returns it back to the client.
- *IN OUT type:*
 - An *INOUT* parameter is passed as a copy with an input value to the Web service method. The Web service method uses the input value, process it, fills in the *INOUT* parameter with a new value and returns it back to the client.



Holder Classes

- WSDL allows for “in/out” parameters to operations
 - parts that appear both in the input and output message
 - service client uses the Holder class instance to send the values of either the out or the in/out parameter.
 - The contents of the Holder class are modified by the remote method calls and the service client can use this changed content after the method invocation.
- in/out parameters are mapped to Holder Classes
- Holder classes implement `javax.xml.rpc.Holders.Holder`



Holder Classes

- Holder Classes for primitive types
 - Named according to the following convention:
 - float's holder is `javax.xml.rpc.holders.FloatHolder`
 - float's holder wrapper is
`javax.xml.rpc.holders.FloatWrapperHolder`
- Holder classes are generated for all other XML types
 - For complex XML data types, the name of the Holder class is constructed by
 - appending Holder to the name of the corresponding Java class
 - example: `com.example.holders.BookHolder`.
 - Holder has a field *value* with the type of the mapped Java class
 - Public constructor that sets *value* to the constructor argument

Holder Classes Example – Sample WSDL

```
<xsd:complexType name="Authors">
  <xsd:all>
    <xsd:element name="Authors" type="typens:AuthorArray"/>
  </xsd:all>
</xsd:complexType>

<message name="AuthorPresentRequest">
  <part name="Authors" type="typens:Authors"/>
</message>
<message name="AuthorPresentResponse">
  <part name="return" type="xsd:boolean"/>
  <part name="Authors" type="typens:Authors"/>
</message>
<portType name="AcmeAuthorPresentPortType">
  <operation name="IsAuthorPresent">
    <input message="typens:AuthorPresentRequest"/>
    <output message="typens:AuthorPresentResponse"/>
  </operation>
</portType>
```

You can see that the input and output messages contain an Authors type as a parameter. This is the in/out style of parameter passing.

Holder Classes Example – Mapped Java

Holder class Definition

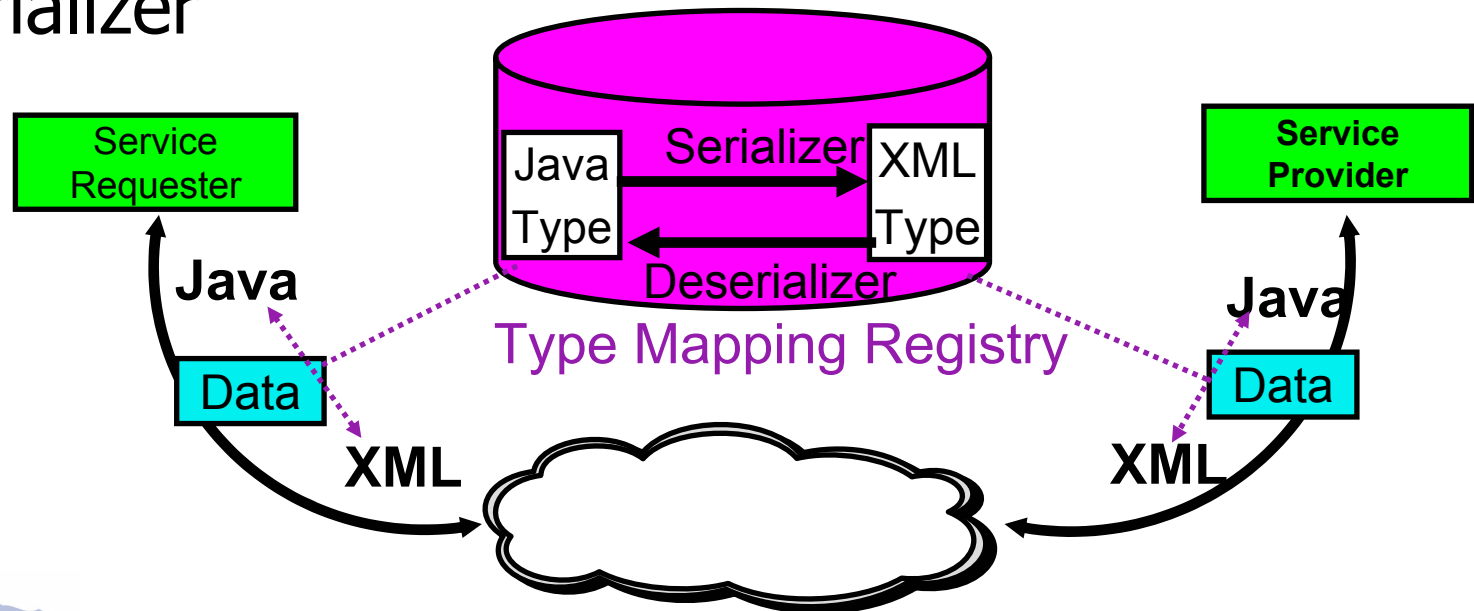
```
public final class AuthorsHolder implements javax.xml.rpc.holders.Holder {  
    public com.acme.www.Authors value;  
    public AuthorsHolder() {}  
    public AuthorsHolder(com.acme.www.Authors value) {  
        this.value = value; }  
}
```

Java Service Endpoint Interface

```
public interface AcmeAuthorPresentPortType extends java.rmi.Remote {  
    public boolean isAuthorPresent(AuthorsHolder authors) throws  
        java.rmi.RemoteException;  
}
```

JAX-RPC Type Mapping Framework

- TypeMappingRegistry
- TypeMapping
- Serializer
- Deserializer



Type Mapping / Custom Serialization

- JAX-RPC defines interfaces which a vendor may provide implementations for to support custom serializers
- JAX-RPC does not specify the XML processing model to be used
- A vendor would extend the Serializer and Deserializer interfaces to support a specific XML processing mechanism (ex. SAX)

Type Mapping / Custom Serialization

- **TypeMappingRegistry** interface is how you lookup type mapping for a Java class or XML type
- **TypeMapping**
 - Maps between
 - `JavaType (Class)`
 - `XML Type (QName)`
 - `javax.xml.rpc.encoding.SerializerFactory`
 - `javax.xml.rpc.encoding.DeserializerFactory`
- **Serializer**
 - Implements `javax.xml.rpc.encoding.Serializer`
- **Deserializer**
 - Implements `javax.xml.rpc.encoding.Deserializer`
- **Get the registry for a service from**
 - `getTypeMappingRegistry()` on `javax.xml.rpc.Service`

SOAP with Attachments

- MIME Types are mapped to Java types according to WSDL MIME binding
- To create an attachment, pass one of the mapped Java types as a parameter or return value
- You may also pass `javax.activation.DataHandler` to handle types not included in the JAX-RPC mapping (implementation dependent).

MIME Type	Java Type
image/gif	<code>java.awt.Image</code>
image/jpeg	<code>java.awt.Image</code>
text/plain	<code>java.lang.String</code>
multipart/*	<code>javax.mail.internet.MimeMultipart</code>
text/xml application/xml	<code>javax.xml.transform.source</code>

Exception Handling

- JAX-RPC specification addresses Web Service run-time (application and system) exceptions
 - Based on the standard approach to service endpoint interface design and its mapping to `wsdl:fault` elements
 - service-specific exceptions are declared in the `wsdl:fault` element, and these exception types are derived from the `java.lang.Exception` class

```
<wsdl:portType name="Transfer_SEI">
  <wsdl:operation name="transferFunds" parameterOrder="fromAcctId toAcctId
amount">
    <wsdl:input name="transferFundsRequest"
        message="impl:transferFundsRequest"/>
    <wsdl:output name="transferFundsResponse"
        message="impl:transferFundsResponse"/>
    <wsdl:fault name="InsufficientFundsException"
        message="impl:InsufficientFundsException" />
  </wsdl:operation>
</wsdl:portType>
```

JAX-B (Java API for XML Binding) JSR31

- Java data binding facility that compiles an XML schema into one or more Java classes
- Interface oriented
 - binding compiler that binds components of a *source schema* to schema-derived Java *content classes*
 - establish conventions for annotating classes with the necessary metadata.
 - standard way to customize the binding of existing schema's components to Java
- Binding framework – APIs for
 - *unmarshalling* of an XML document into a tree of interrelated instances of both existing and schema-derived classes,
 - *marshalling* of such *content trees* back into XML
 - *validation* of content trees against the constraints expressed in the schema.



JAX-RPC and JAXB

- According to the JAX-RPC 1.1 specification draft
 - You may or may not use JAXB for for converting Java to XML
 - JAXB does however have some impact on type mapping
 - In order to make it possible to use other binding frameworks in JAX-RPC
 - It must be possible to selectively turn the standard JAX-RPC mapping off on a per part basis
 - JAX-RPC tools are required to provide a facility for specifying metadata for this purpose



JAX-RPC Handlers

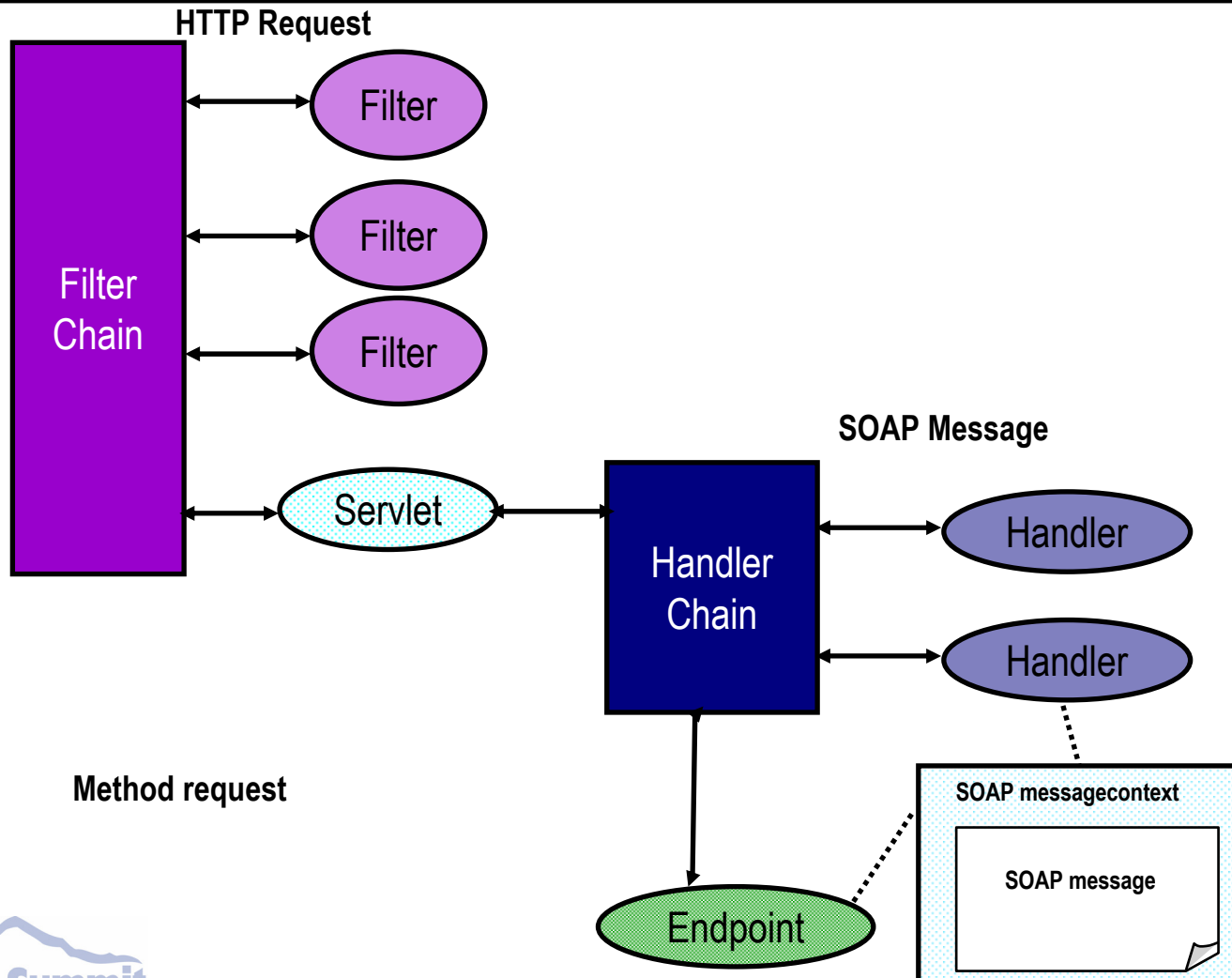


JAX-RPC Handlers

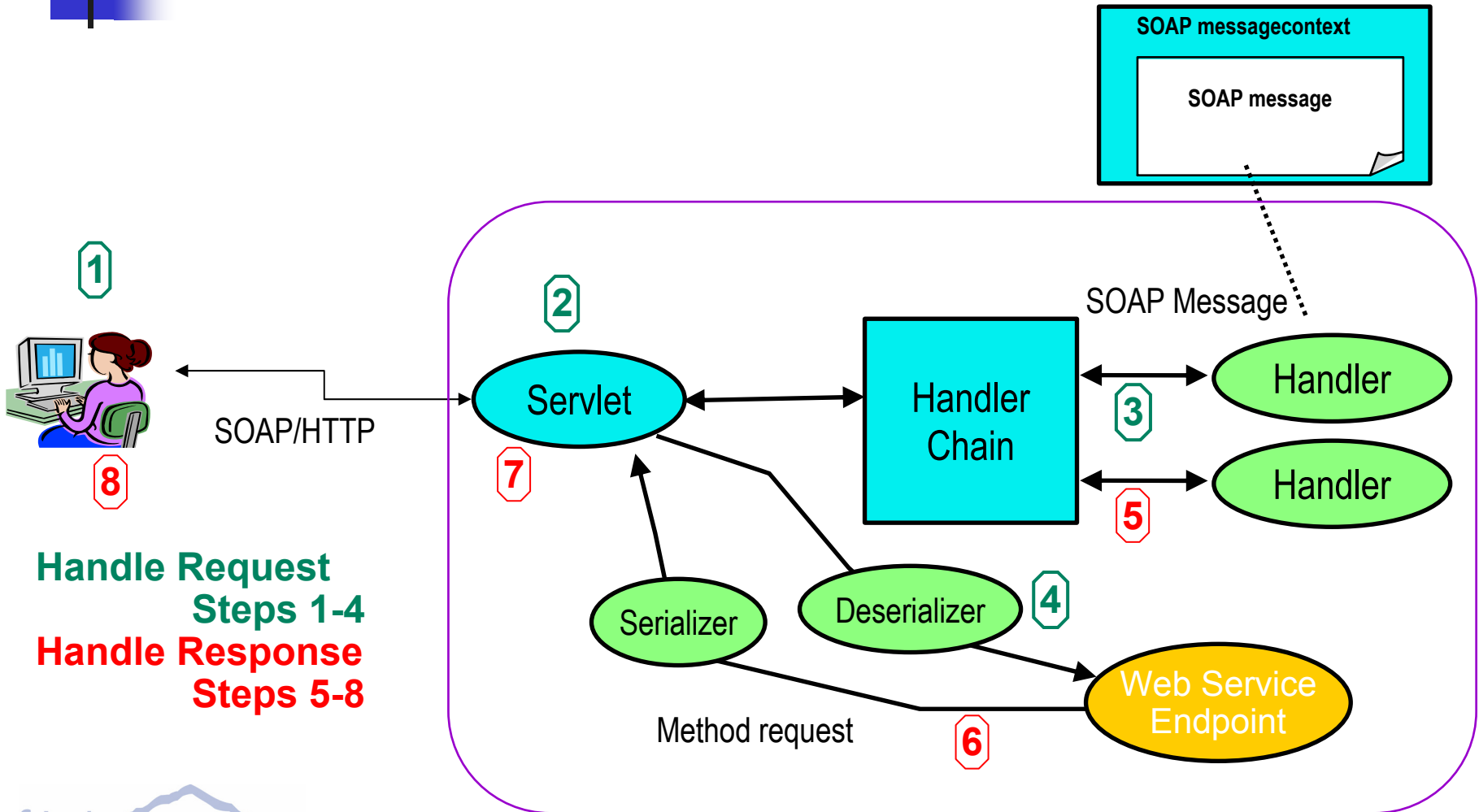
- Provides a mechanism for intercepting the SOAP message and operating on header information
 - Can examine and potentially modify a request before it is processed by a Web Service component.
 - Can examine and potentially modify the response after the component has processed the request
- Conceptually, similar to Servlet 2.3 Filter
- Can be provided for both the client and server
- Handler **MUST NOT CHANGE SOAP message, Operation name, number of parts in the message, or types of the message parts**
 - SOAP Fault is send if Handler does this
- Handlers are configured into 'ordered chains'
 - Client side Handlers run after the Stub/proxy has marshaled the message, but before container services and the transport binding occurs.
 - Server side Handlers run after container services have run including method level authorization, but before demarshalling and dispatching the SOAP message to the endpoint.

Handler chains (handlers) are service specific

JAX RPC Handler Architecture

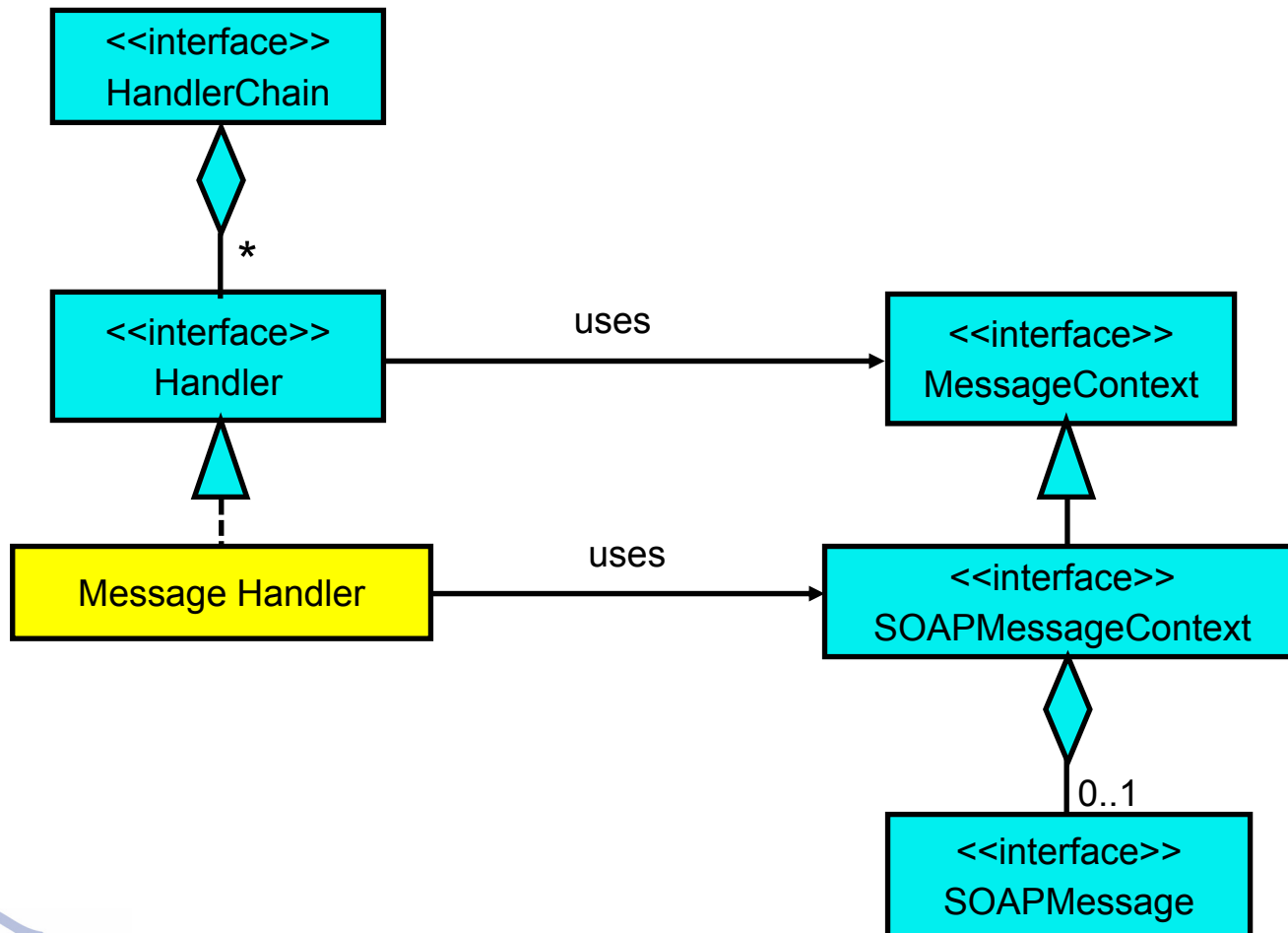


End to End Flow for SOAP/HTTP



Handle Request
Steps 1-4
Handle Response
Steps 5-8

Handler Class Hierarchy





Handler Configuration

- Handlers are configured programmatically *via* `HandlerRegistry`
 - The `HandlerRegistry` is available *via* the `getHandlerRegistry` method on `Service`
 - `HandlerRegistry` has methods to
 - `getHandlerChain`
 - `setHandlerChain`
- J2EE Deployment Descriptor for Handlers (JSR109)



Handler Interface

```
package javax.xml.rpc.handler;  
public interface Handler(){  
    boolean handleRequest(MessageContext ctx);  
    boolean handleResponse(MessageContext ctx);  
    boolean handleFault(MessageContext ctx);  
}
```



Header Handling

- The `handleRequest()` method does the following
 - Return `true` to indicate that the processing of the handler chain should continue
 - Return `false` to indicate that the handler chain is blocked.
 - Processing continues in on the `handleResponse()` method of this handler instance and the handlers backward in the chain
 - Throw a `JAXRPCException` or other `RuntimeException` for a runtime error (in which case the `HandlerChain` will generate a SOAP Fault)
- The `handleResponse()` mechanism is similar, except for the processing false case
 - After blocking (return false), no other handlers will be processed



Handler Chains

- A Handler is associated with a SOAP header block using the qualified name of the outermost element of each header block
- A Handler chain performs the following steps during SOAP processing
 - Identify the set of SOAP actor roles in which this handler chains is to act
 - Identify all mandatory header blocks for this role
 - If one or more of the mandatory blocks are not handled by this node then generate a SOAP MustUnderstand fault
 - Otherwise, process all the header blocks by invoking the chain of handlers
 - If processing is unsuccessful, generate exactly one SOAP fault to propagate to the client



Handler Caveats

- Usually need to be defined in pairs (client, server)
- Client cannot communicate requirements to server handler and vice-versa
 - Not described in WSDL
- Client cannot alter the signature, hence, can't change the operation or type
- Customers need to consider performance issues
 - Cost associated with each Handler

JSR109 – Enterprise Services Programming Model



JSR-109 Implementing Enterprise Web Services

- Key Web Services Objective
 - Achieve interoperability across heterogeneous platforms and runtimes
- Basically filling the gaps that the JAX-RPC specification left open for use of Web services in a J2EE Application Server
- JSR 109 - Web Services for J2EE
 - Facilitates the building of interoperable Web Services in J2EE
 - Standardization of the the deployment of Web Services in a J2EE container
 - Specifically addresses
 - Client Access to Web Services
 - Web Service Lifecycle
 - Web Service Deployment

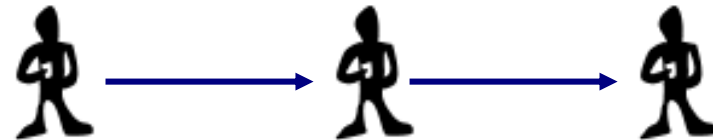


J2EE Roles

- J2EE Roles
 - J2EE Product Provider
 - Application Component Provider
 - Developer
 - Application Assembler
 - Deployer
 - System Administrator
 - Tool Provider

J2EE Web Services Roles

- Integrating Web Services into J2EE adds additional responsibilities to to the J2EE defined roles
 - Application Component Provider
 - Developer
 - Application Assembler
 - Deployer



Developer

Assembler

Deployer



JSR-109 Developer

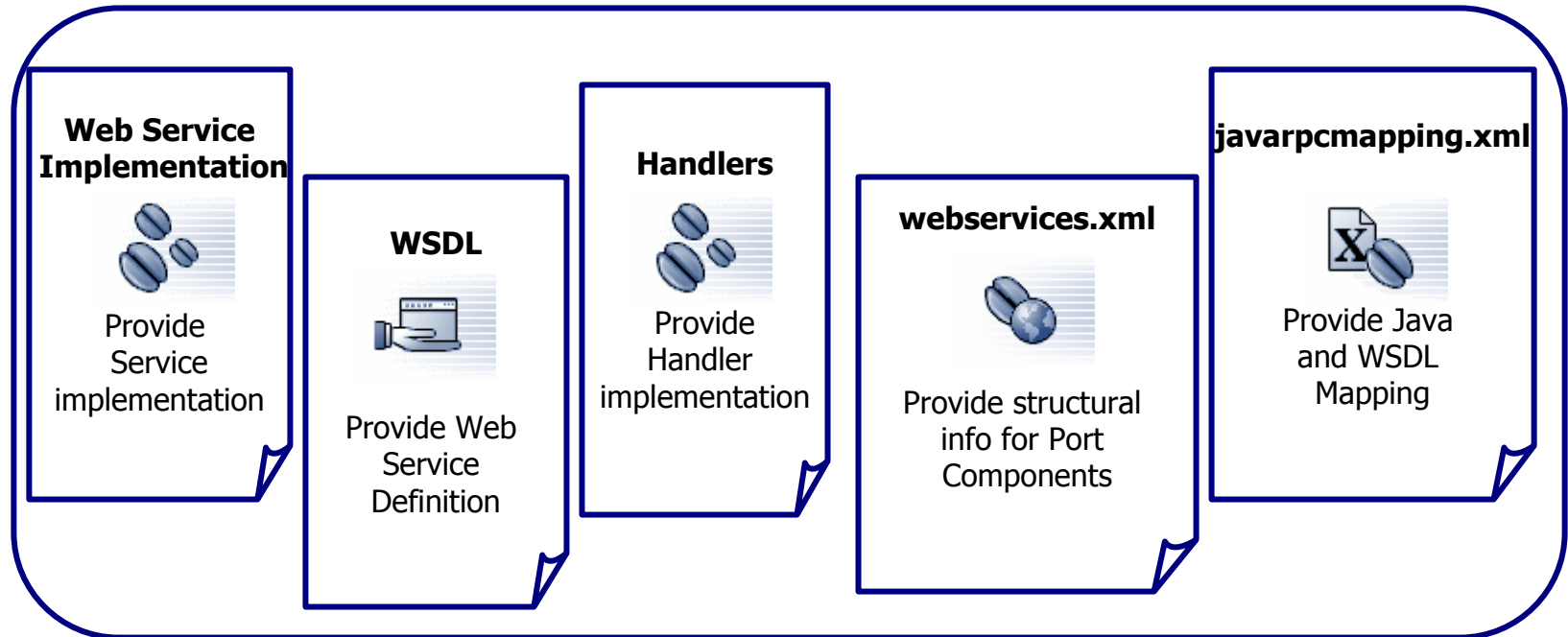
- Developer is responsible for providing
 - Web Service definition
 - Implementation of the Web Service
 - Structural information for the Web Service
 - Implementation of handlers
 - Java and WSDL mappings
 - Packaging of Web Service related artifacts into a J2EE Module



Developer

JSR-109 Developer

Server Programming Model



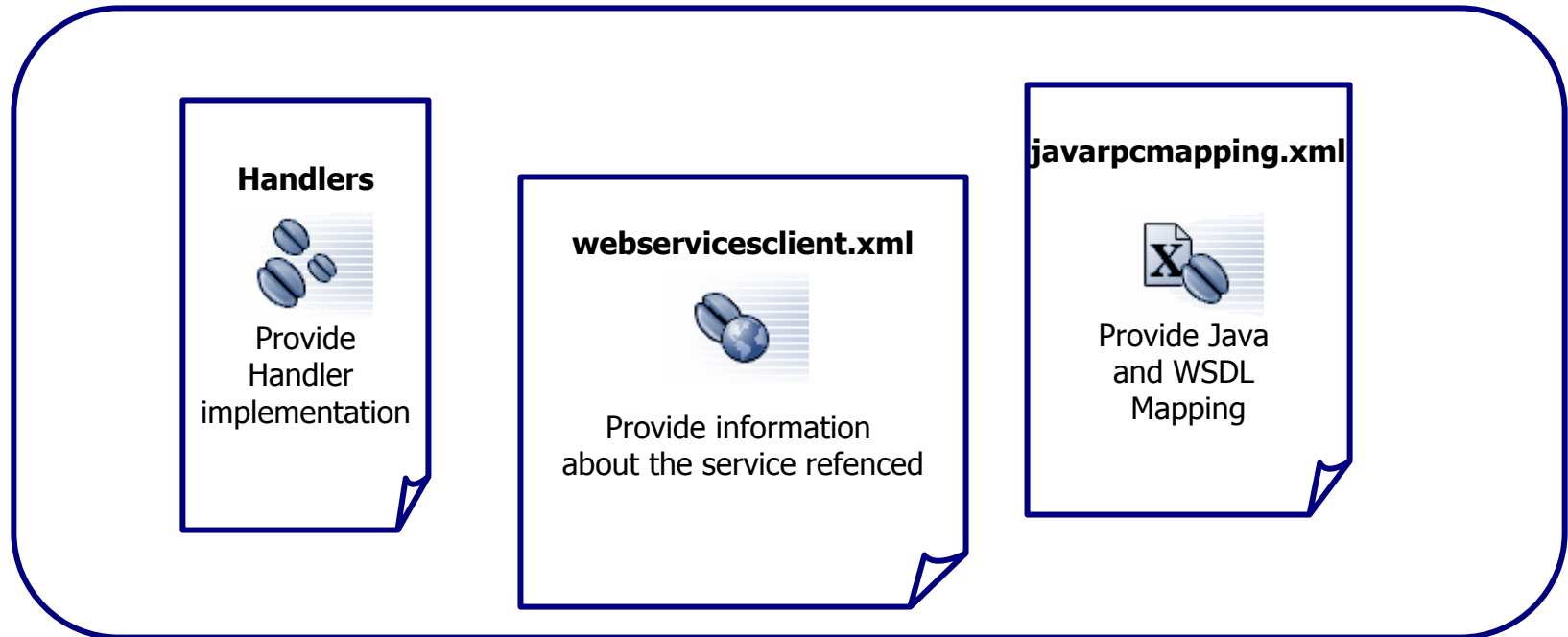
J2EE Module



Developer

JSR-109 Developer

Client Programming Model



J2EE Module



Developer



JSR-109 Assembler

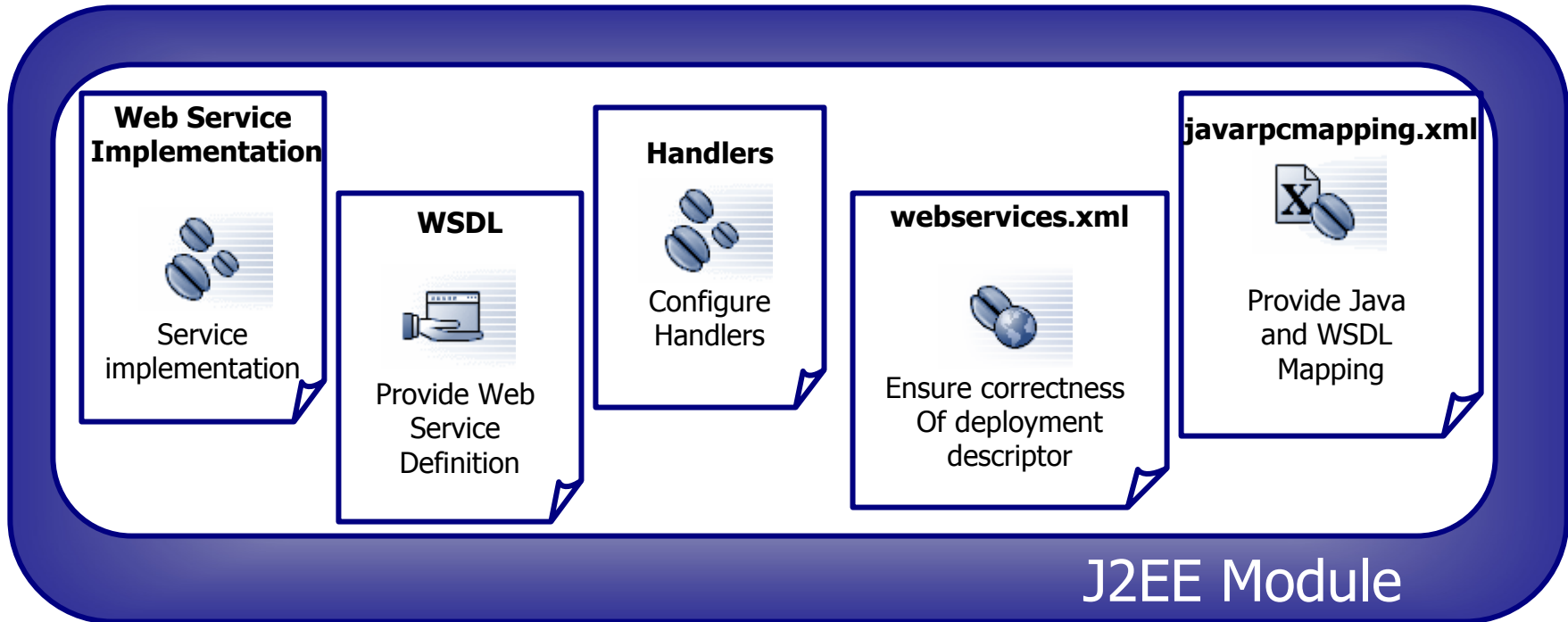
- Assembler is responsible for
 - Assembling modules into and Enterprise Application Archive (EAR)
 - Configuring the modules within the application
 - Module dependencies
 - Configuring handlers



Assembler

JSR-109 Assembler

Server Programming Model



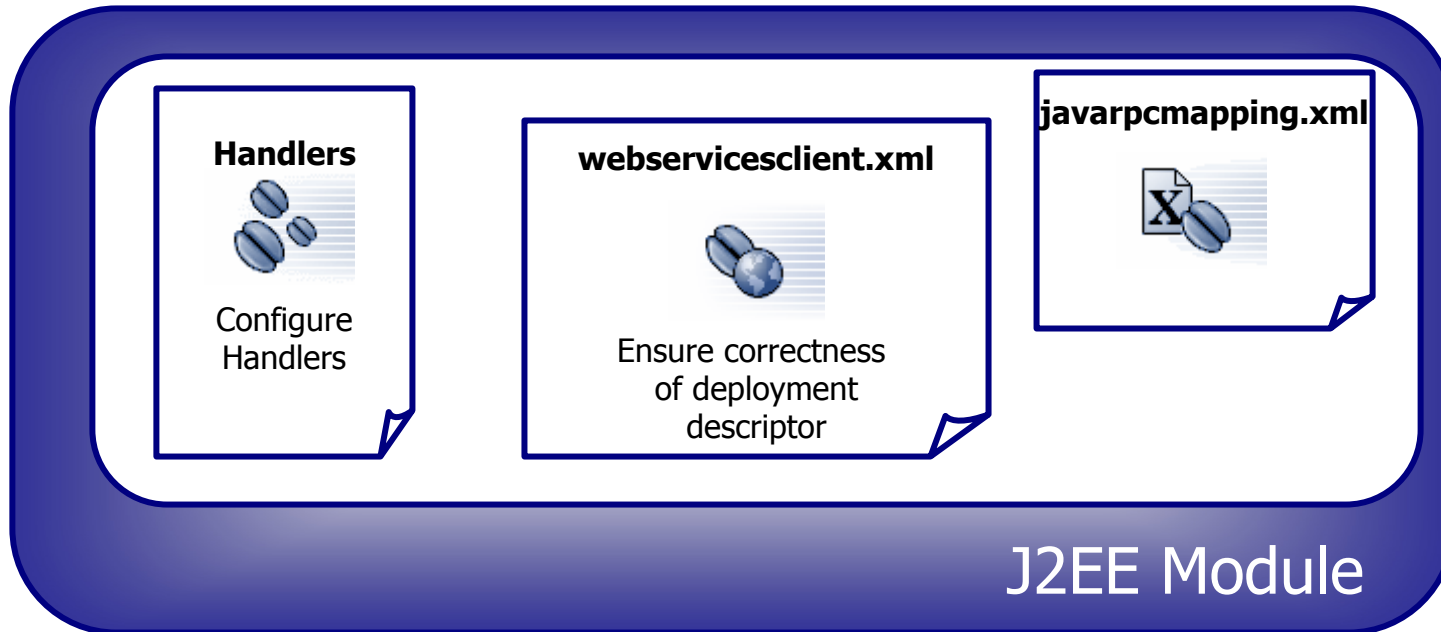
Enterprise Application



Assembler

JSR-109 Assembler

Client Programming Model



Enterprise Application



Assembler

JSR-109 Deployer

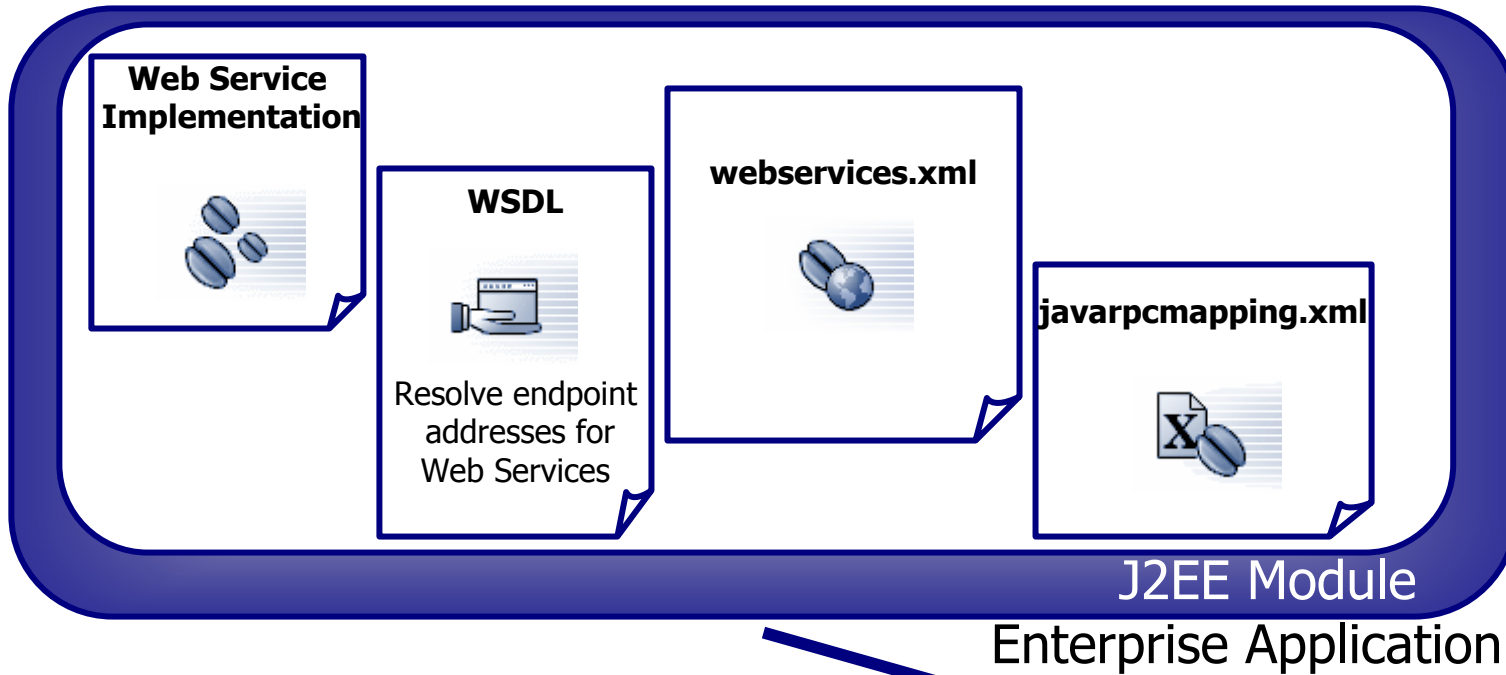
- Deployer is responsible for
 - Resolving endpoint addresses fro Web Services contained in the EAR
 - Generating stubs for Web Services
 - using deployment tools provided by the Web Services for J2EE (JSR-109) provider
 - Resolving the WSDL documents for each service reference
 - Resolving port access declared by a port component reference
 - Deploying the enterprise application



Deployer

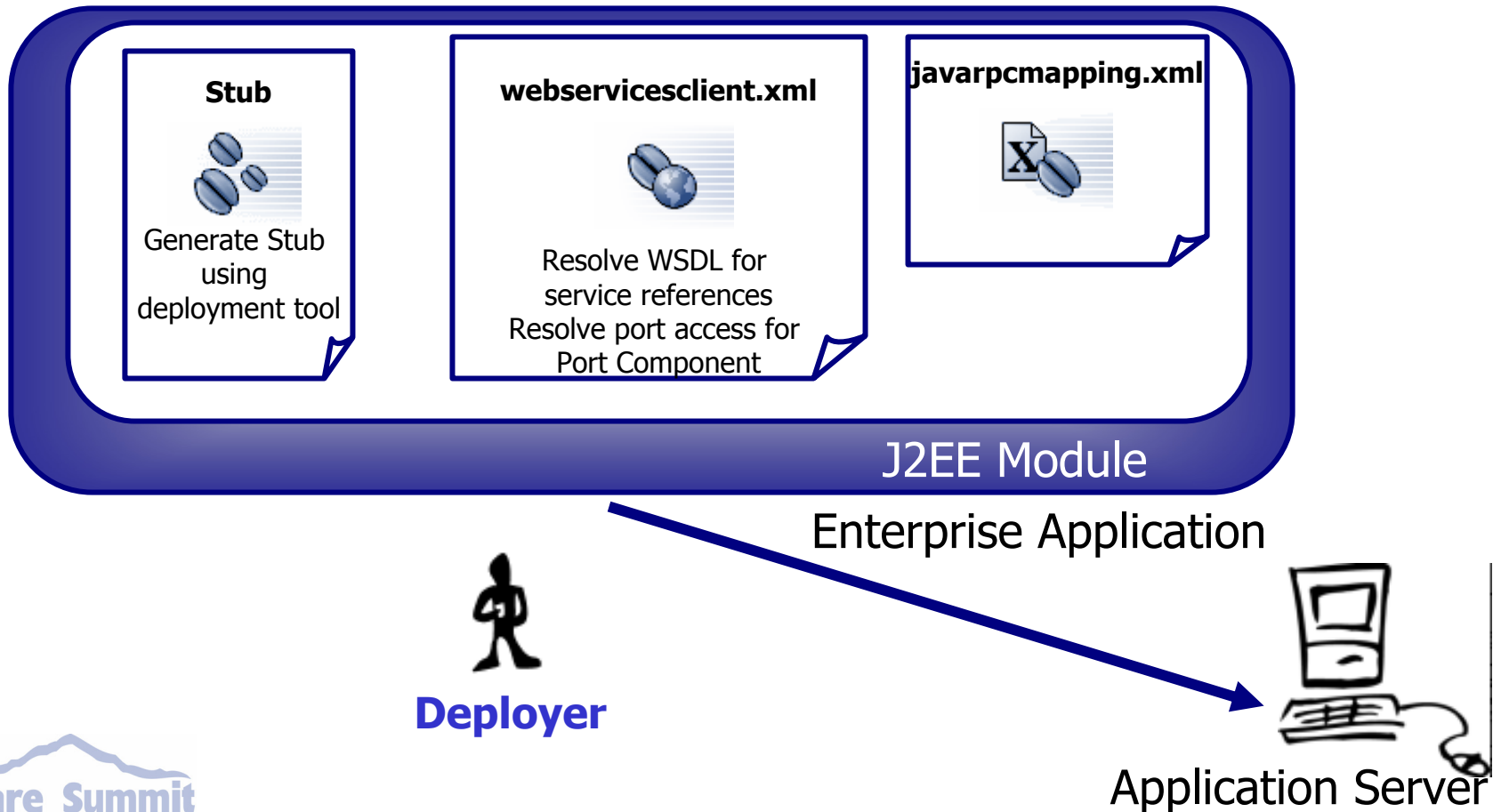
JSR-109 Deployer

Server Programming Model



JSR-109 Assembler

Client Programming Model





Service Programming Models



JSR-109 Server Programming Model

- Attempts to standardize the deployment of Web Services
- Defines deployment of
 - Servlet based implementation bean in a Web Container
 - Stateless Session EJB implementation in an EJB container
- Use of the Service Endpoint Interface (SEI) defined by JAX-RPC
 - Defines the method signatures of the Web Service
 - Must follow the Java and WSDL mapping rules defined by JAX-RPC

JSR-109 Service Bean Implementation

- The Service Implementation Bean must
 - Implement all the methods defined by the SEI
 - For EJBs
 - SEI methods must be a subset of the remote interface methods
- The application server (WebApp for HTTP) must route incoming SOAP requests to the appropriate Service Bean Implementation
 - New deployment descriptors are defined
- The Service Implementation Bean and state
 - The container may create and pool multiple instances of the bean to optimize request handling
 - Thus carrying state may be questionable and risky

JSR-109 Service Bean Implementation

- WSDL is used as the contract for the Web Service
- The WSDL to SEI mapping **MUST** follow the Java and WSDL mapping rules defined by JAX-RPC



Service LifeCycle

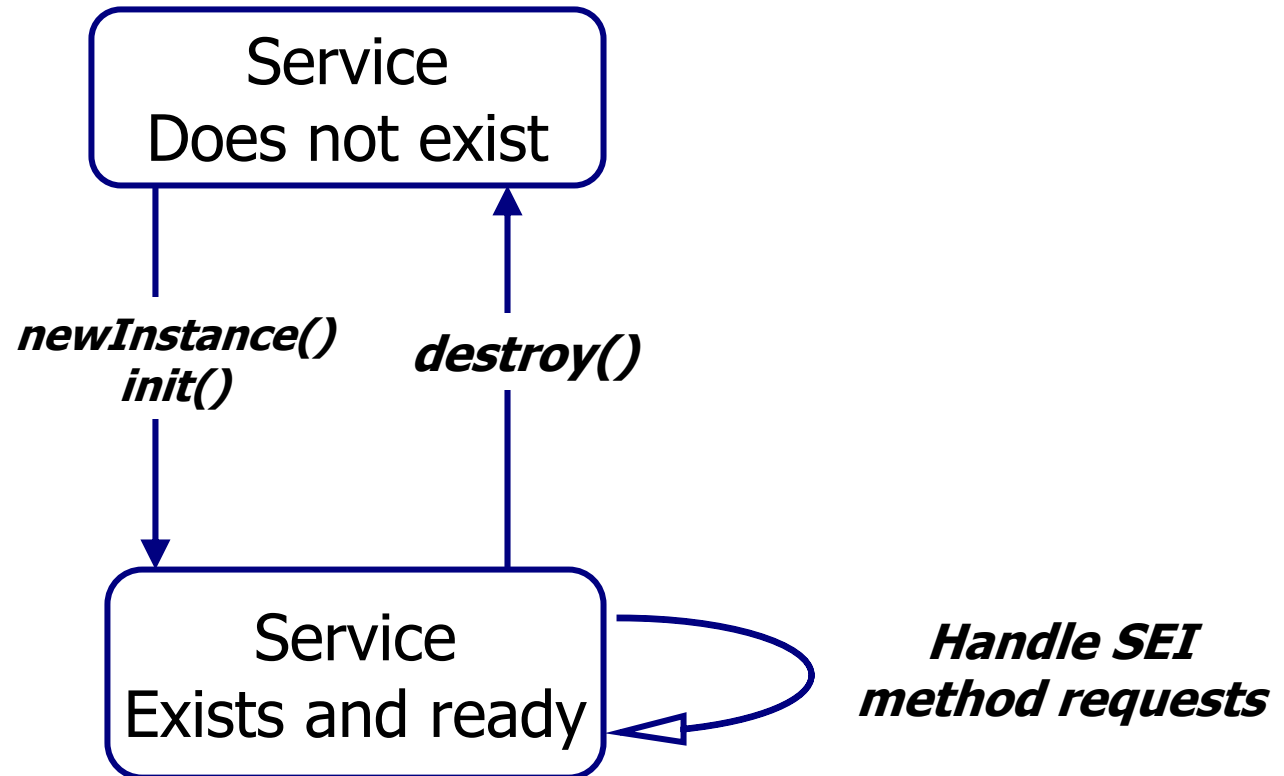
- LifeCycle is controlled by the associated container
- In general, life cycle phases are
 - Instantiation
 - Initialization
 - Execution
 - Removal



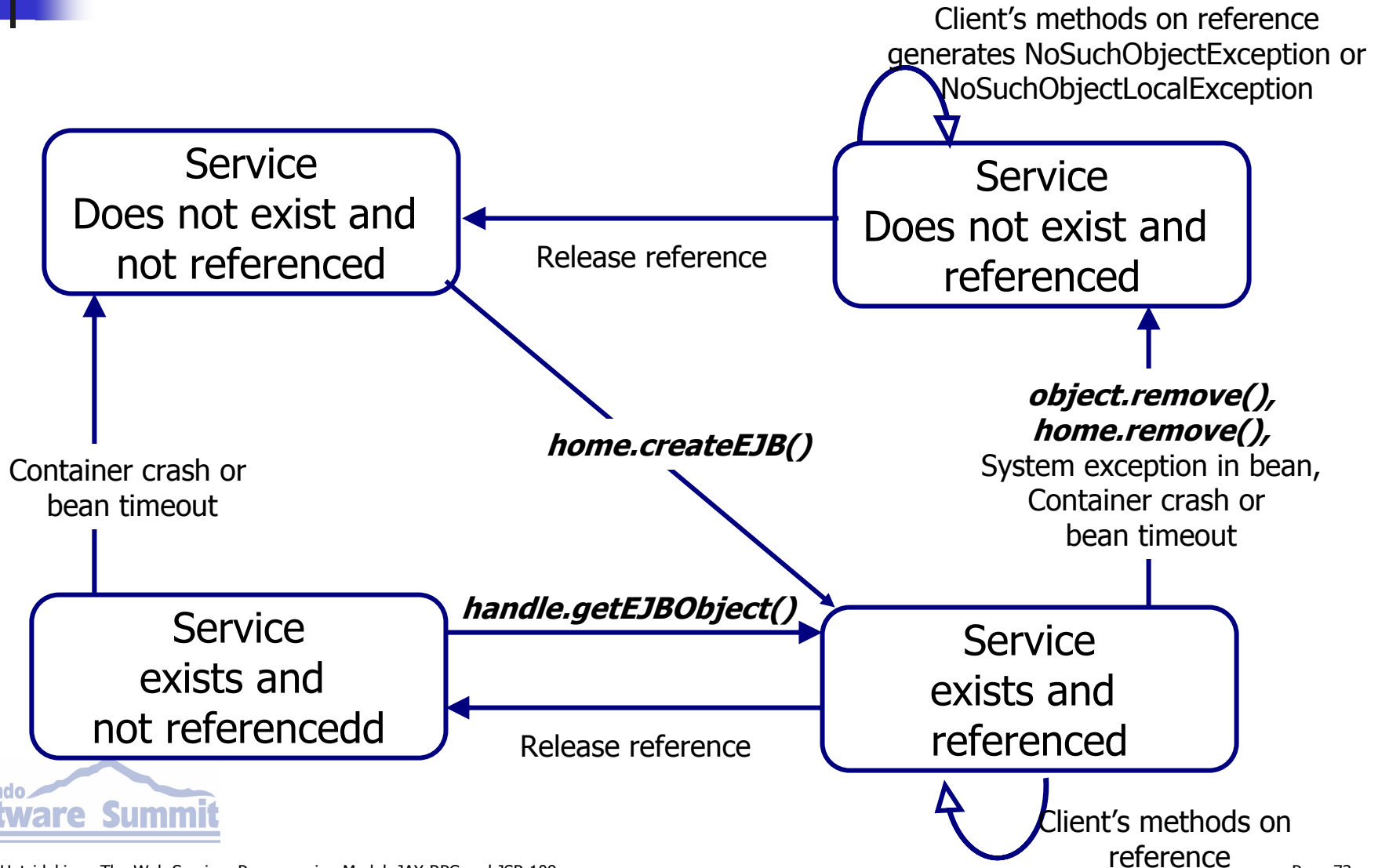
Web Service LifeCycle

- Instantiation
 - Container invokes the bean's *newInstance()* method
- Initialization
 - Container initializes *via* the container specific method
 - Servlet based service – *init()*
 - EJB based service – *setSessionContext()*, *ejbCreate()*
- Execution
- Removal
 - Container removes the service
 - Servlet based service – *destroy()*
 - EJB based service – *ejbRemove()*

Bean Based Implementation



EJB Based Implementation Bean





Server Side Packaging

- Port components (SEI and Implementation)
 - In their respective modules – Web or EJB

- Web Services DD – webservices.xml
 - For EJB, in the Module META-INF directory
 - For Web, in the Module WEB-INF directory

- WSDL and Mapping files
 - Located relative to the module
 - Typically co-located with the module descriptor
 - Referenced in the Deployment Descriptor



Service Deployment Descriptors

- Specify the set of Web service descriptions to be deployed and their dependencies on container resources and services
- Specify any necessary type mappings
- Specify the Java artifact (JavaBean / EJB) implementing the service



Client Programming Models



Types of Clients

- JAX-RPC Client (Standalone Java Client)
 - Defined by JSR 101, not defined by JSR 109
 - Uses the JSR-101 client programming model
 - WSDL definition of a Web Service provides enough information for a non-Web Services J2EE client to be built and run, but the programming model for that is undefined
- Web Services for J2EE Client
 - Defined by JSR 109
 - Runs in a J2EE Container and uses J2EE run-time to access/invoke the methods of a Web Service
 - Can be
 - J2EE Application client
 - Web component (Servlet/JSP)
 - EJB component

Types of Client Access to Service

- **Client-Managed Access** (Standalone Java Client)
 - Lookup of service and stub defined by JAX-RPC (not defined by JSR 109)
 - Does not require a J2EE container
 - The client code must know fully-specified URL of the WSDL and therefore the Web service address/endpoint
- **Container-Managed Access** (Web Services for J2EE Client)
 - Lookup of SEI and stub is defined by JSR 109
 - Runs in a J2EE Container and uses J2EE run-time to access/invoke the methods of a Web Service
 - The client code does not know the URL of the target Web service
 - It only has either the SEI or the Port Type and bindings part of the WSDL
 - Can be

● J2EE Application client

● Web component (Servlet/JSP)

● EJB component



JSR-109 Client Programming Model

- Provides a client view of the Web service in a J2EE environment
- Runtime details are transparent to the client
 - Protocol binding
 - Transport
- A Web Service is invoked much like invoking a method locally



Client Model Invocation Styles

- Three invocation styles:
 - Static (generated) stub invocation
 - Java class is statically bound to an SEI, WSDL Port and port component
 - Pretty much as with existing Web Services Implementations
 - Dynamic Proxy invocation
 - Java Class is not statically bound to the Web Service *via* a generated Stub
 - A dynamic proxy can be obtained at runtime by providing the SEI
 - Uses the JDK 1.3 Dynamic Proxy feature
 - Dynamic Call DII invocation
 - `javax.xml.rpc.Call` object is instantiated and configured to invoke the Web service
 - Used when a client needs dynamic, non-stub based communication with the Web Service



Client Program Flow

- Get Service
- Get stub from the service
- Instantiate and setup the objects which are parameters to the operation
- Use the stub to invoke the remote service operation
- Process return message or fault

JAX-RPC Client Using a Static Stub

- Uses the Service Factory class to create instance of a `javax.xml.rpc.Service` class

```
Service addressBookService = ServiceFactory.newInstance().createService(  
    new URL("file", "", " Addressbook.wsdl "),  
    new QName("http://addr", "AddressBookService"));
```

- On the Service class, invoke `getPort` with the Port name, to return an instance of a stub that implements the SEI interface
 - The stub will be generated during deployment and are vendor-specific
 - The stub is the client representation of an instance of the Web Service

```
AddressBook stub = (AddressBook) addressBookService.getPort(  
    new QName("http://addr", "AddressBook"), AddressBook.class);
```

- Client uses the stub to drive the Web Service request to the endpoint that implements the Web service

```
Address response = stub.getAddress();
```



JAX-RPC Client Running in J2EE Container

- Client uses JNDI lookup that returns a Service object
- Using the Service Object, the client gets the stub (that implements the Service Interface)

```
InitialContext initCtx = new InitialContext();  
AddressBookService addressBookService = (AddressBookService)  
    initCtx.lookup("java:com/env/service/AddressBook");  
AddressBook stub =  
addressBookService.getPort(AddressBook.class);
```

- Client uses the stub to drive the Web Service request
Address response = stub.getAddress();



Dynamic Invocation Model

- Implement `javax.xml.rpc.Call`

```
Service service = ServiceFactory.newInstance().createService(null);
```

```
Call call = service.createCall();
```

```
URL u = new URL(
```

```
    "http://localhost:6080/AddressBookService/services/AddressBook");
```

```
call.setTargetEndpointAddress(u);
```

```
call.setProperty(Call.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
```

```
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "getAddress");
```

```
call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
```

```
    "http://schemas.xmlsoap.org/soap/encoding/");
```

```
call.setOperationName(new QName("urn:address-book", "getQuote"));
```

```
call.addParameter("name", XMLType.XSD_STRING, ParameterMode.IN);
```

```
call.setReturnType("urn:address", AddressBook.class);
```

```
Object result = call.invoke(new Object[] {name = "John Doe"});
```

```
return (Address) result;
```

Choosing the Client API to Use

- **Static**
 - These are generated by the tooling
 - Use when you know the service location is unlikely to change
 - You want compile time type checking of use of stub
- **Dynamic Proxy**
 - Dynamic Proxy allows you to use a service endpoint interface without a generated stub class
 - Program to the static SEI, but can change the location of the service endpoint
 - The SEI allows you to retrieve a port and pass in the endpoint URL
- **DII**
 - Use when you need to figure out the service location / signature dynamically
 - Allows late/dynamic binding
 - No service endpoint interface is required

What about Service Discovery?

- Remote Service Discovery must be done by developer
 - Not defined by JSR-109 or JAX-RPC
 - Locate and retrieve WSDL *via*:
 - URL
 - UDDI
 - UDDI4J
 - JAX-R
- From the WSDL pick one of:
 - Service and Port
 - Service and PortType
- Can then use the JAX-RPC client-managed programming model to connect to a particular Web service implementation

Client



JAX-RPC Runtime Services

- Provided *via* property interface on Stub and Call interfaces

- HTTP Basic Authentication
 - `javax.xml.rpc.security.auth.username` (String)
 - `javax.xml.rpc.security.auth.password` (String)
 - No other HTTP authentication forms are required

- HTTP Session Management
 - `javax.xml.rpc.session.maintain` (boolean)

Command Line Tools – Java2WSDL

- Creates WSDL document from Java classes
- Command: `java2wsdl [options] class`
- Some important Options
 - location:
 - provides the service location within WSDL
 - style
 - RPC | DOCUMENT
 - use
 - ENCODED | LITERAL
 - transport
 - http | jms
 - `implClass <class> :`
 - uses method parameter names from impl-class to construct the WSDL message part names

Command Line Tools – WSDL2Java

- Creates Java artifacts and/or Web Services Deployment Descriptor templates from WSDL
- Command
 - WSDL2Java [*options...*] wsdl-file
- Some important Options
 - role j2ee-role, where j2ee-role are
 - **develop-client** (default): generates files for client development
 - **develop-server**: generates files for server development
 - **deploy-client**: generates binding files for client deployment
 - **deploy-server**: generates binding files for server deployment
 - **client**: combination of **develop-client** and **deploy-client**
 - **server**: combination of **develop-server** and **deploy-server**
 - container j2ee-container, where j2ee-container is:
 - **none**: indicates no container
 - **client**: indicates client container
 - **ejb**: indicates EJB container
 - **web**: indicates Web container
 - genJava argument: Generates Java files, where arguments are
 - No
 - IfNotExists (default for develop roles)
 - Overwrite (default for deployment roles)



JSR109 Deployment Descriptors

Deployment Descriptors

- Service Deployment Descriptor
 - `webservices.xml`
 - Packaged in same directory as module deployment descriptor
 - `META-INF` for EJB and Application client modules
 - `WEB-INF` for WAR modules
 - Stateless Session Bean Service Implementation defined by `session` element in `ejb-jar.xml`
 - Web container Service Implementation defined by `servlet-class` element in `web.xml`
- Client Deployment Descriptor
 - `webservicesclient.xml`
 - Packaged in same directory as module deployment descriptor
 - `META-INF` for EJB and Application client modules
 - `WEB-INF` for WAR modules
- JAX-RPC Mapping Deployment Descriptor
 - `<service>_mapping.xml`
 - Packaged in same directory as module deployment descriptor
 - `META-INF` for EJB and Application client modules
 - `WEB-INF` for WAR modules

JSR-109 Service Deployment Descriptors

- JSR-109 builds on top of JAX-RPC and defines two deployment descriptors
 - webservices.xml
 - Defines the structural information of the deployed Web services
 - Each WSDL defines service is mapped to a description in webservices.xml
 - JAX-RPC mapping file
 - The name of this file is not dictated by the specification
 - Referenced by the webservices.xml

WSDL and webservicexml

```

<wsdl:service name="StockQuoteService">
  <wsdl:port binding="intf:GetQuoteBinding" name="GetQuote">
    <wsdlsoap:address location="http://www.example.com/stockquote/services/getquote"/>
  </wsdl:port>
  <wsdl:port binding="intf:PurchaseBinding" name="Purchase">
    <wsdlsoap:address location="http://www.example.com/stockquote/services/purchase"/>
  </wsdl:port>
</wsdl:service>
<wsdl:service name="ExchangeRateService">
  <wsdl:port binding="intf:ExchangeRateBinding" name="ExchangeRate">
    <wsdlsoap:address location="http://www.example.com/exchangerate/services/getrate"/>
  </wsdl:port>
</wsdl:service>

```

WSDL

```

<webservicexml>
  <webservice-description>
    <webservice-description-name>StockQuoteService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
    ...
  </webservice-description>
  <webservice-description>
    <webservice-description-name>ExchangeRateService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
    ...
  </webservice-description>
</webservicexml>

```

webservicexml



webservices.xml

- Each description element **MUST** contain
 - Name
 - References the service in the WSDL file
 - WSDL file reference
 - References the WSDL file location in the J2EE module
 - JAX-RPC mapping file reference
 - References the Mapping file in the J2EE module



webservices.xml

```
<webservices>
  <webservice-description>
    <webservice-description-name>ExchangeRateService</webservice-description-name>
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
    <port-component>
      <port-component-name>ExchangeRate</port-component-name>
      <wsdl-port>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>ExchangeRate</localpart>
      </wsdl-port>
      <service-endpoint-interface>sample.ExchangeRatePortType</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>sample_ExchangeRateBindingImpl</servlet-link>
      </service-impl-bean>
    </port-component>
  </webservice-description>
  ...
</webservices>
```



webservices.xml – Ports

- Port
 - Defines the server view of the service
- Contains
 - Name
 - WSDL port reference
 - Namespace URI
 - WSDL local name of the wsdl:port
 - SEI reference
 - Fully qualified class name of the SEI
 - SIB reference
 - Web Service implementation
 - ✓ Servlet based service – references a servlet element in the web.xml
 - ✓ EJB based service – references an EJB defined in ejb-jar.xml

webservices.xml – Port component

```

<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  targetNamespace="http://sample"
  .....
```

.....

```

  <wsdl:service name="StockQuoteService">
    <wsdl:port binding="intf:GetQuoteBinding" name="GetQuote">
      <wsdlsoap:address location="http://www.example.com/stockquote/services/getquote"/>
    </wsdl:port>
    <wsdl:port binding="intf:PurchaseBinding" name="Purchase">
      <wsdlsoap:address location="http://www.example.com/stockquote/services/purchase"/>
    </wsdl:port>
  </wsdl:service>
  .....
```

.....

```

</wsdl:definitions>

```

WSDL

```

<webservice-description>
  <webservice-description-name>StockQuoteService</webservice-description-name>
  <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
  <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
  <port-component>
    <port-component-name>Purchase</port-component-name>
    <wsdl-port>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>Purchase</localpart>
    </wsdl-port>
    .....
```

.....

```

  </port-component>
  .....
```

```

</webservice-description>

```

webservices.xml

Servlet Based Service Port

```
<servlet>
  <servlet-name>sample_GetQuoteBindingImpl</servlet-name>
  <servlet-class>sample.GetQuoteBindingImpl</servlet-class>
</servlet>
```

web.xml

```
<webservices>
  <webservice-description>
    ...
    <port-component>
      <port-component-name>GetQuote</port-component-name>
      <wsdl-port>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>GetQuote</localpart>
      </wsdl-port>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
      <service-impl-bean>
        <servlet-link>sample_GetQuoteBindingImpl</servlet-link>
      </service-impl-bean>
    ...
  </webservice-description>
</webservices>
```

webservices.xml

EJB Based Service Port

```

<enterprise-beans>
  <session>
    <ejb-name>GetQuoteBindingImpl</ejb-name>
    <home>sample.GetQuotePortTypeHome</home>
    <remote>sample.GetQuotePortType_RI</remote>
    <ejb-class>sample.GetQuoteBindingImpl</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</trab=nsaction-type>
  </session>
</enterprise-beans>

```

ejb-jar.xml

```

<webservices>
  <webservice-description>
    ...
    <port-component>
      <port-component-name>GetQuote</port-component-name>
      <wsdl-port>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>GetQuote</localpart>
      </wsdl-port>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
      <service-impl-bean>
        <ejb-link>GetQuoteBindingImpl</ejb-link>
      </service-impl-bean>
    ...
  </webservice-description>
</webservices>

```

webservices.xml



Handlers

- JSR-109 also standardizes the support for JAX-RPC handlers
 - Handlers can
 - Pre-process a request before it is dispatched to a Web Service endpoint
 - Post-process a response before it is sent to the client
 - Access and modify the SOAP Header and content if the SOAP binding protocol is used
 - Handler **MUST NOT CHANGE SOAP message structure, Operation name, number of parts in the message, or types of the message parts**
- SOAP Fault is send if Handler does this



Handlers

```

<port-component>
  <port-component-name>GetQuote</port-component-name>
  <wsdl-port>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuote</localpart>
  </wsdl-port>
  <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-
interface>
  <service-impl-bean>
    <servlet-link>sample_GetQuoteBindingImpl</servlet-link>
  </service-impl-bean>
  <handler>
    <handler-name>sample.ValidationHandler</handler-name>
    <handler-class>sample.ValidationHandler</handler-class>
  </handler>
  <handler>
    <handler-name>sample.LoggingHandler</handler-name>
    <handler-class>sample.LoggingHandler</handler-class>
    <init-param>
      <param-name>level</param-name>
      <param-value>warning</param-value>
      <description>Logging level</description>
    </init-param>
    <soap-header>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>GetQuote</localpart>
    </soap-header>
    <soap-role>LoggingHandler</soap-role>
  </handler>
</port-component>

```

**Validation
Handler**

**Logging
Handler**

webservices.xml

JSR-109 Client Deployment Descriptors

- Similar to service descriptors – two deployment descriptors
 - `webservicesclient.xml`
 - Defines each Web Service referenced by the client
 - `jaxrpcmapping.xml`
 - Defines the JAX-RPC mapping used by the client to access the Web Services



webservicesclient.xml

- Each service referenced by the client is defined by a service-ref which contains
 - Name
 - References the service in the WSDL file
 - Service Interface
 - Fully qualified class name of the Service Interface
 - WSDL file reference
 - References the WSDL file location in the J2EE module
 - JAX-RPC mapping file reference
 - References the Mapping file in the J2EE module



webservicesclient.xml

```
<webservicesclient>
  <service-ref>
    <description>WSDL Service ExchangeRateService</description>
    <service-ref-name>service/ExchangeRateService</service-ref-name>
    <service-interface>sample.ExchangeRateService</service-interface>
    <wsdl-file>WEB-INF/wsdl/Sample.wsdl</wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/Sample_mapping.xml</jaxrpc-mapping-file>
    <service-qname>
      <namespaceURI>http://sample</namespaceURI>
      <localpart>ExchangeRateService</localpart>
    </service-qname>
    <port-component-ref>
      <service-endpoint-interface>sample.ExchangeRatePortType</service-endpoint-interface>
    </port-component-ref>
  </service-ref>
  ...
</weervicesclient>
```



Service Interface

- Service Interface
 - A java representation of the Web Service
 - Must implement the `javax.xml.rpc.Service` interface
 - Can be used to generate
 - A Stub class
 - A Dynamic Proxy
 - A `javax.xml.rpc.Call` object

Service Interface and SEI

```
package sample;

public interface StockQuoteService extends javax.xml.rpc.Service
{
    public java.lang.String getPurchaseAddress();
    public sample.PurchasePortType getPurchase()
        throws javax.xml.rpc.ServiceException;
    public sample.PurchasePortType getPurchase(java.net.URL portAddress)
        throws javax.xml.rpc.ServiceException;
    public java.lang.String getQuoteAddress();
    public sample.GetQuotePortType getQuote() throws javax.xml.rpc.ServiceException;
    public sample.GetQuotePortType getQuote(java.net.URL portAddress)
        throws javax.xml.rpc.ServiceException;
}
```

Service Interface

```
package sample;

public interface GetQuotePortType extends javax.rmi.Remote
{
    public sample.Price getQuote(sample.Ticker parameter)
        throws java.rmi.RemoteException;
}
```

Service Endpoint Interface



Client Side Handlers

- Client side handlers are associated with service references
 - Server side handlers are associated with port component references
- They can
 - Process a request before it is sent to the service endpoint
 - Process a response before it is returned to the client
- Defined the same way as Service Handlers
 - Additional the port name parameter
 - Port names are used to associate handlers with WSDL ports



Client Side Handlers

```

<webservicessclient>
  <service-ref>
    ...
    <port-component-ref>
      <service-endpoint-interface>sample.PurchasePortType</service-endpoint-interface>
    </port-component-ref>
    <port-component-ref>
      <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
    </port-component-ref>
    <handler>
      <handler-name>sample.LoggingHandler</handler-name>
      <handler-class>sample.LoggingHandler</handler-class>
      <init-param>
        <param-name>level</param-name>
        <param-value>information</param-value>
        <description>Logging level</description>
      </init-param>
      <soap-header>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>GetQuote</localpart>
      </soap-header>
      <soap-role>LoggingHandler</soap-role>
      <port-name>GetQuote</port-name>
    </handler>
  </service-ref> ...
</webservicessclient>

```



JAX-RPC Mapping

- Mechanism for standardizing the Java ↔ WSDL mappings
- JAX-RPC provides the rules for the mappings
- JSR-109 provides the XML-based deployment descriptor standardized representation
- No standardized name
 - Name determined by the `jaxrpc-mapping-file` element in `webservices.xml` or `webservicesclient.xml`
- File structure matches closely to the WSDL file structure



JAX-RPC Mapping

- `<package-mapping>` element
 - Defines Java package and namespace mapping

```
<java-wsdl-mapping>
  <package-mapping>
    <package-type>sample</package-type>
    <namespaceURI>http://sample</namespaceURI>
  </package-mapping>
  ...
</java-wsdl-mapping>
```

WSDL Service ↔ Service Interface Mapping

```

<wsdl:service name="StockQuoteService">
  <wsdl:port binding="intf:GetQuoteBinding" name="GetQuote">
    <wsdlsoap:address
      location="http://www.example.com/stockquote/services/getquote"/>
    </wsdl:port>
  <wsdl:port binding="intf:PurchaseBinding" name="Purchase">
    <wsdlsoap:address
      location="http://www.example.com/stockquote/services/purchase"/>
    </wsdl:port>
</wsdl:service>

```

WSDL

```

<service-interface-mapping>
  <service-interface>sample.StockQuoteService</service-interface>
  <wsdl-service-name>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>StockQuoteService</localpart>
  </wsdl-service-name>
  <port-mapping>
    <port-name>Purchase</port-name>
    <java-port-name>Purchase</java-port-name>
  </port-mapping>
  <port-mapping>
    <port-name>GetQuote</port-name>
    <java-port-name>GetQuote</java-port-name>
  </port-mapping>
</service-interface-mapping>

```

JAX-RPC Mapping file

WSDL Mappings

```

<wsdl:message name="getQuoteRequest">
  <wsdl:part element="intf:ticker" name="parameter"/>
</wsdl:message>
<wsdl:message name="getQuoteResponse">
  <wsdl:part element="intf:price" name="parameter"/>
</wsdl:message>
...
<wsdl:portType name="GetQuotePortType">
  <wsdl:operation name="getQuote">
    <wsdl:input message="intf:getQuoteRequest" name="getQuoteRequest"/>
    <wsdl:output message="intf:getQuoteResponse" name="getQuoteResponse"/>
  </wsdl:operation>
</wsdl:portType>
...
<wsdl:binding name="GetQuoteBinding" type="intf:GetQuotePortType">
  <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="getQuote">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="getQuoteRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="getQuoteResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

```

WSDL

WSDL Bindings Mappings

```

<service-endpoint-interface-mapping>
  <service-endpoint-interface>sample.GetQuotePortType</service-endpoint-interface>
  <wsdl-port-type>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuotePortType</localpart>
  </wsdl-port-type>
  <wsdl-binding>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>GetQuoteBinding</localpart>
  </wsdl-binding>
  <service-endpoint-method-mapping>
    <java-method-name>getQuote</java-method-name>
    <wsdl-operation>getQuote</wsdl-operation>
    <method-param-parts-mapping>
      <param-position>0</param-position>
      <param-type>sample.Ticker</param-type>
      <wsdl-message-mapping>
        <wsdl-message>
          <namespaceURI>http://sample</namespaceURI>
          <localpart>getQuoteRequest</localpart>
        </wsdl-message>
        <wsdl-message-part-name>parameter</wsdl-message-part-name>
        <parameter-mode>IN</parameter-mode>
      </wsdl-message-mapping>
    </method-param-parts-mapping>
    <wsdl-return-value-mapping>
      <method-return-value>sample.Price</method-return-value>
      <wsdl-message>
        <namespaceURI>http://sample</namespaceURI>
        <localpart>getQuoteResponse</localpart>
      </wsdl-message>
      <wsdl-message-part-name>parameter</wsdl-message-part-name>
    </wsdl-return-value-mapping>
  </service-endpoint-method-mapping>
</service-endpoint-interface-mapping>

```

Java ↔ XML Mapping

```

<wsdl:types>
  <schema
    elementFormDefault="qualified"
    targetNamespace="http://sample"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="ticker">
      <complexType>
        <sequence>
          <element name="item" type="xsd:string"/>
        </sequence>
      </complexType>
    ...
  </wsdl:types>

```

WSDL

```

<java-xml-type-mapping>
  <class-type>sample.Ticker</class-type>
  <root-type-qname>
    <namespaceURI>http://sample</namespaceURI>
    <localpart>&gt;ticker</localpart>
  </root-type-qname>
  <qname-scope>complexType</qname-scope>
  <variable-mapping>
    <java-variable-name>item</java-variable-name>
    <xml-element-name>item</xml-element-name>
  </variable-mapping>
</java-xml-type-mapping>
<java-xml-type-mapping>
  <class-type>java.lang.String</class-type>
  <root-type-qname>
    <namespaceURI>http://www.w3.org/2001/XMLSchema</namespaceURI>
    <localpart>string</localpart>
  </root-type-qname>
  <qname-scope>simpleType</qname-scope>
</java-xml-type-mapping>

```

JAX-RPC Mapping file



Backup



Migration from Apache SOAP

Differences Between JAX-RPC and the Apache Models

Feature	JAX-RPC	Apache model
Java Bean Service Implementations	Service Implementations must implement an Interface the extends <code>java.rmi.Remote</code>	No Interface required
RPC Parameters	Complex Types (Beans) used must implement <code>java.io.Serializable</code> . All public fields must be non-transient.	No implementation of <code>Serializable</code> required
Document/Literal style SOAP	Is interpreted as specific method parameters maps where possible, else uses a <code>SOAPElement</code>	A common method parameter list taking in the raw XML is used
Can access the HTTP Session in the Web Service	Yes, through the <code>ServletEndpointContext</code>	Not available in the standard API



Migration Strategy

- Use WSDL
 - Generate client stubs for use by your application
- Use an Isolation layer
 - Don't talk directly to the Apache SOAP API
 - Create a layer that your application uses to talk to Apache SOAP



Basic Migration

- Apache SOAP's Call API looks like the JAX-RPC DII
 - Method names have changed slightly
 - Parameter passing is slightly different
 - Apache SOAP uses a Vector of Parameters
 - JAX-RPC uses multiple calls to addParameter and an object array
 - JAX-RPC value types must implement java.io.Serializable
- Accessing runtime services is different
 - Apache SOAP uses methods on SOAPHTTPConnection
 - JAX-RPC uses setProperty on the Stub or Call object

Basic Migration: Type mapping

- XML Types not mapped by JAX-RPC but mapped by Apache SOAP
 - `xsd:timeInstant`
 - `xsd:date`
- Java Types not mapped by JAX-RPC
 - `java.util.GregorianCalendar`
 - `java.util.Date`
 - `java.util.Vector`
 - `java.util.Enumeration`
 - `java.util.Hashtable`
 - `java.util.Map`
- Conflicting types
 - `org.apache.soap.util.xml.QName` vs `javax.xml.namespace.QName`
- JavaBean type mapping
 - `org.apache.soap.encoding.soapenc.BeanSerializer` vs JAX-RPC value types

Advanced Migration: Custom Serialization

■ Custom serialization

- You need to modify your Apache SOAP custom serializers to use the JAX-RPC interfaces
 - `javax.xml.rpc.encoding.Serializer`
 - `javax.xml.rpc.encoding.Deserializer`
- The details of the JAX-RPC custom serializers interfaces is implementation dependent
- You also need to provide
 - an implementor of `javax.rpc.encoding.SerializerFactory`
 - an implementor of `javax.rpc.encoding.DeserializerFactory`



Advanced Migration

- SOAP with Attachments

- Apache SOAP uses it's own datatype for dealing with attachments, which is not interoperable
- Apache SOAP only creates attachments for arguments of type `javax.activation.DataHandler`.
- JAX-RPC may be more specific and use one of `java.awt.Image`, `java.lang.String`, `javax.mail.internet.MimeMultipart`, `javax.xml.transform.Source`

- Header processing

- Apache SOAP header processing and JAX-RPC Handlers are completely different.
- Apache SOAP uses the DOM to manipulate the Message/Headers
- JAX-RPC uses SAAJ/JAXM's SOAPMessage classes



Server/Container Migration

■ Services

- Best is to start with WSDL for the service and generate JSR-109 deployment descriptors and service stubs
- Either
 - Cut and paste old service into new service stubs
 - Have new service stubs be an Adaptor for old service code
- Trouble areas
 - literal encoding
 - SOAP with Attachments
 - Handlers
 - Custom Serialization

■ Deployment descriptors

- If you followed the recommendation above, you're done
- Else
 - XSLT stylesheet to convert WSDD to webservices.xml
 - Editing required