Web Services Development & Deployment

Srikanth Patil-esolutions (KANA)

Agenda

After the End of the Session you should be familiar with

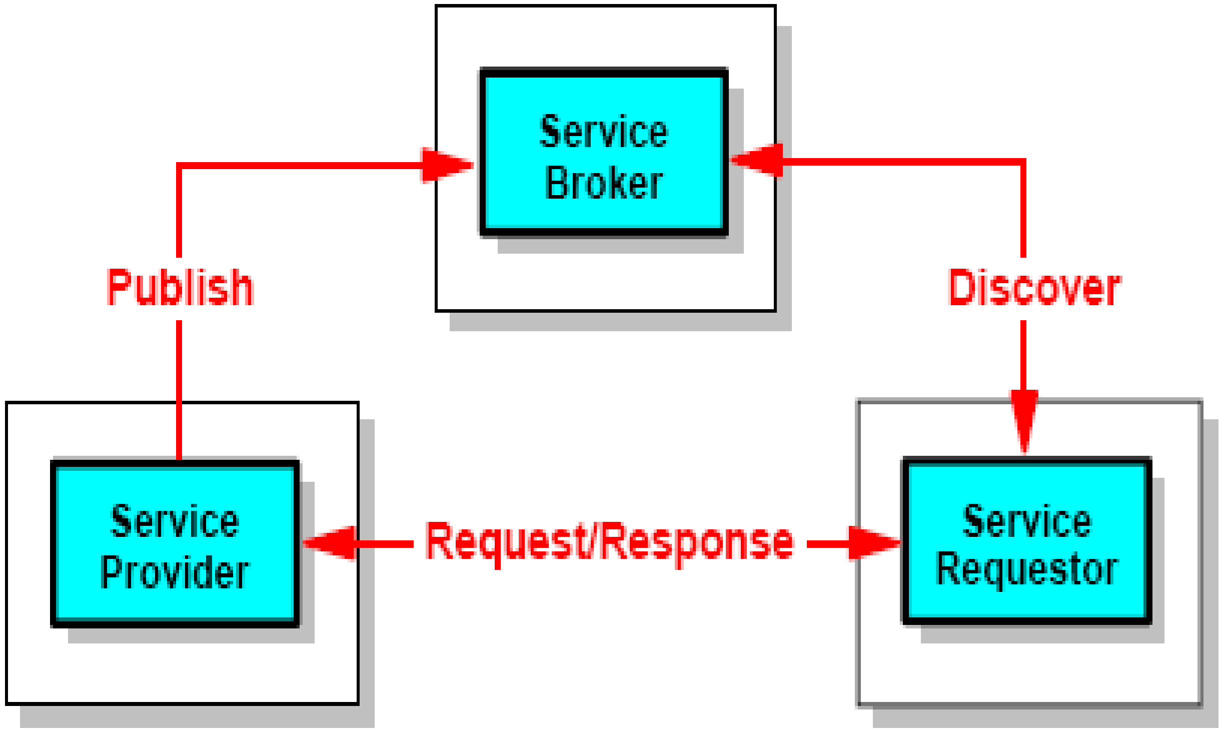
» Introduction to SOA ,Web services & SOAP
» WSDL
» UDDI & WSIL
» Developing the Web services using RAD/WSAD

Service-oriented architecture

A service-oriented architecture consists of three basic components:

- ❑ Service provider
- ❑ Service broker
- ❑ Service requestor

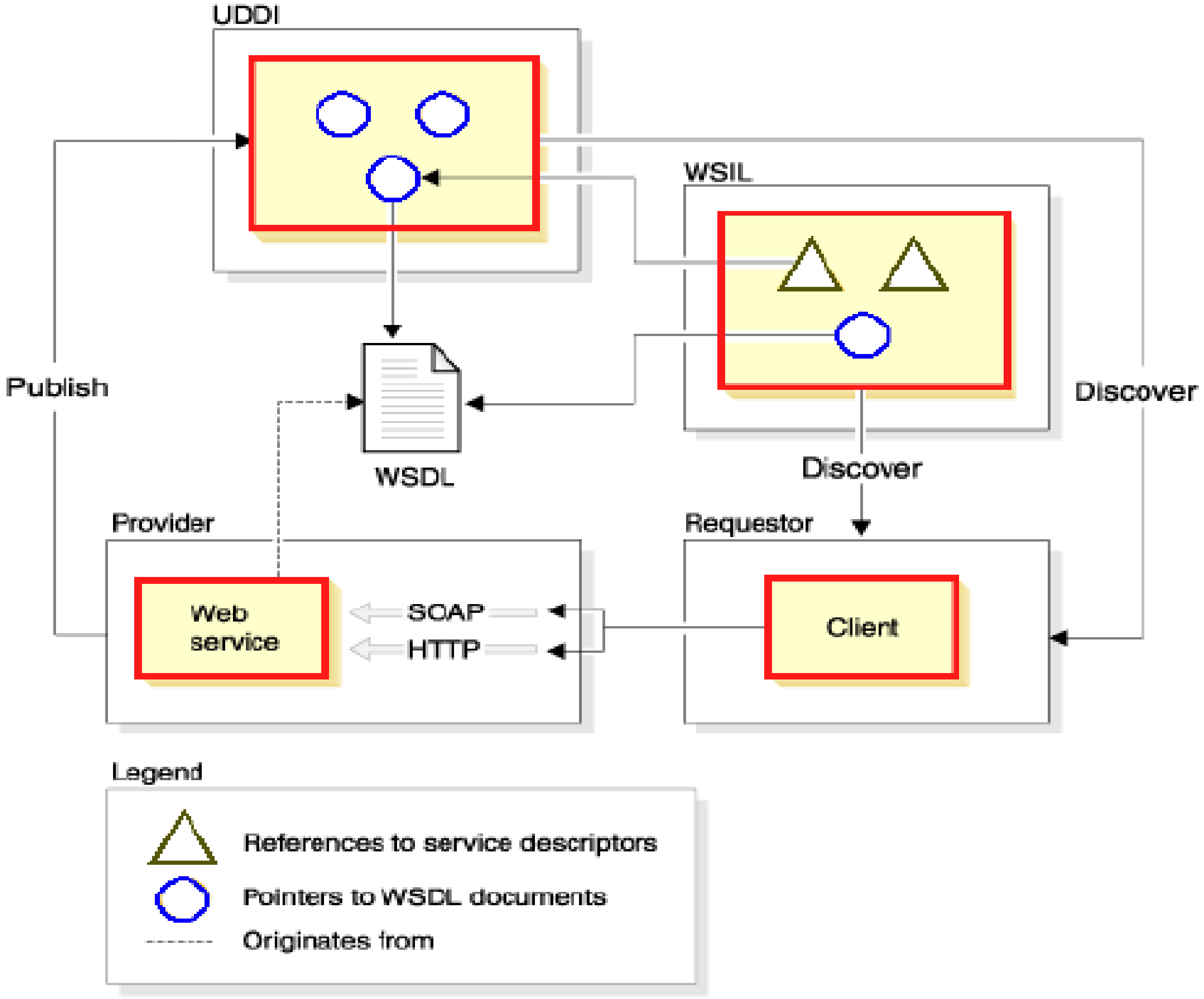
Web services components and operations



Web services components and operations

- The service provider creates a Web service and possibly publishes its interface and access information to the service broker
- The service broker (also known as service registry) is responsible for making the Web service interface and implementation access information available to any potential service requestor.
- The service requestor locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its Web services.
- Web services are self-contained, modular applications that can be described,published, located, and invoked over a network

Main building blocks in an SOA approach based on Web services



Core technologies used for Web services

XML

Is the markup language that underlies most of the specifications used for Web services.

SOAP

(Simple Object Access Protocol) is a network, transport, and programming language and platform-neutral protocol that allows a client to call a remote service. The message format is XML.

WSDL

(Web Services Description Language) is an XML-based interface and implementation description language. The service provider uses a WSDL document in order to specify the operations a Web service provides and the parameters and data types of these operations. A WSDL document also contains the service access information.

Core technologies used for Web services

WSIL

(Web Services Inspection Language) is an XML-based specification about how to locate Web services without the necessity of using UDDI. However, WSIL can be also used together with UDDI, that is, it is orthogonal to UDDI and does not replace it.

UDDI

(Universal Description, Discovery, and Integration) is both a client-side API and a SOAP-based server implementation that can be used to store and retrieve information on service providers and Web services.

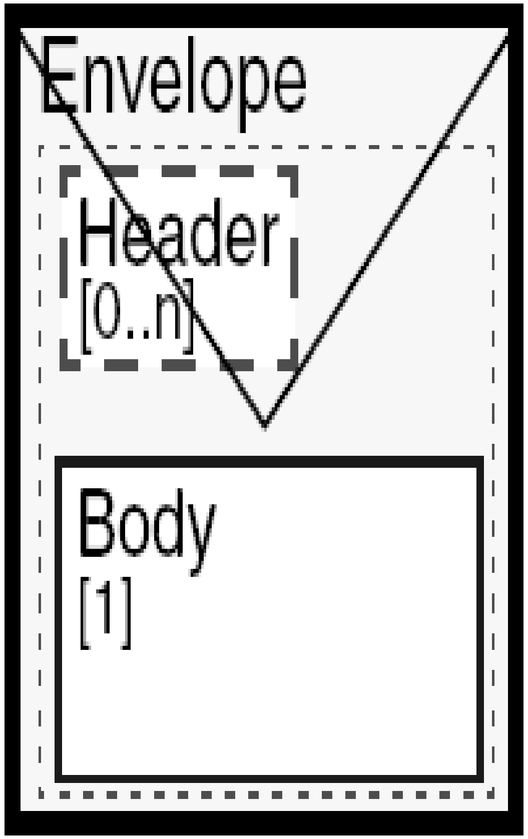
SOAP

- Simple Object Access Protocol (SOAP) is a specification for the exchange of structured information in a decentralized, distributed environment. As such, it represents the main way of communication between the three main actors in SOA: The service provider, service requestor, and service broker.
- SOAP is an XML-based protocol that consists of three parts: An envelope that defines a framework for describing message content and process instructions, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses.
- The way SOAP applications communicate when exchanging messages is often referred to as the message exchange pattern (MEP). The communication can be either one-way messaging, where the SOAP message only goes in one direction, or two-way messaging, where the receiver is expected to send back a reply.

The three pillars of SOAP

- A SOAP message is an envelope containing zero or more headers and exactly one body
- The envelope is the top element of the XML document, providing a container for control information, the addressee of a message, and the message itself.
- Headers contain control information, such as quality of service attributes.
- The body contains the message identification and its parameters.
- Both the headers and the body are child elements of the envelope.

Example of a conceptualized SOAP message



```
<Envelope>
  <Header>
    <actor>http://...org/soap/actor/next</actor>
    <qos mustUnderstand="1">log</qos>
  </Header>
  <Body>
    <getMessage ns1="urn:NextMessage" ...>
    <UserID type="string">JDoe</UserID>
    <Password type="string">0JDOE0</Password>
    </getMessage>
  </Body>
</Envelope>
```

SOAP BODY

- The soap body element contains the content of the message
- The Body element MUST be namespace qualified and reside in the same namespace as that used for the Envelope in which it resides
- The soap body may contain any number of child elements
- The encoding Style attribute may be used to identify the method used to encode the body content
- If the body contains a Fault element , it should not contain any other child elements .

The three pillars of SOAP

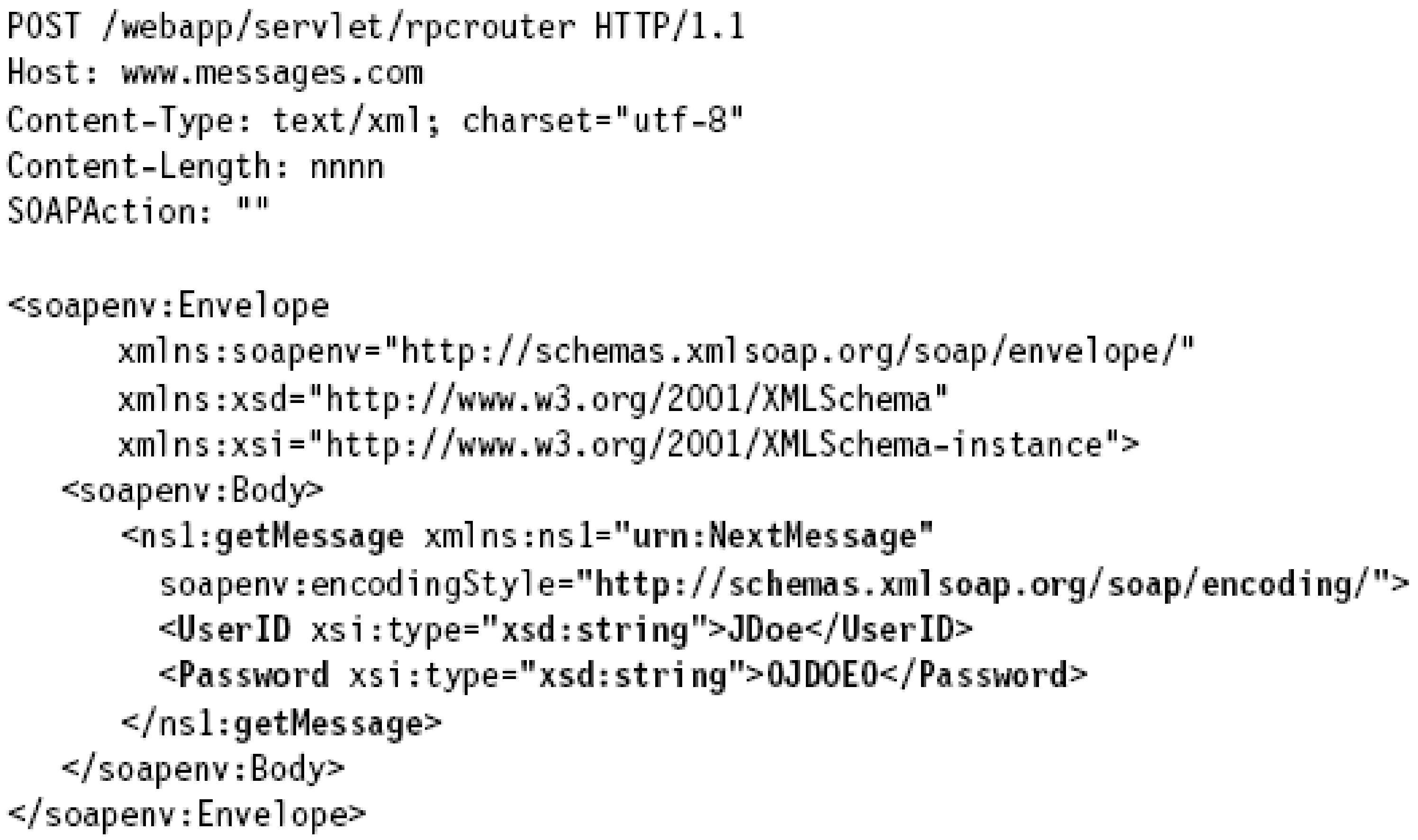
Encoding rules

Encoding rules (of course included in a real SOAP message) define a serialization mechanism that can be used to exchange instances of application-defined data types**.**

RPC representation

The remote procedure call (RPC) representation is a convention suited to represent remote procedure calls and the related response messages. As arguments in remote method invocation, we normally use relatively simple data structures, although, with conventions such as XML Literal, it is possible to transfer more complex data.

A SOAP message embedded in an HTTP request



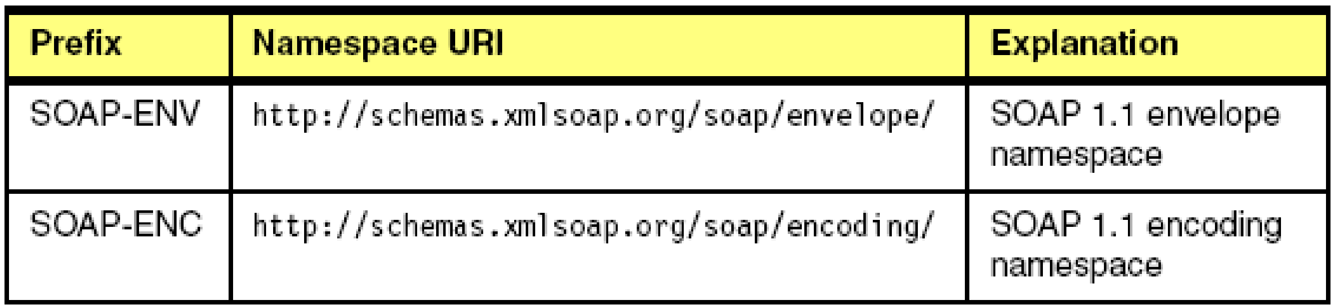
```
POST /webapp/servlet/rpcrouter HTTP/1.1
Host: www.messages.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<soapenv:Envelope
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <soapenv:Body>
      <ns1:getMessage xmlns:ns1="urn:NextMessage"
        soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <UserID xsi:type="xsd:string">JDoe</UserID>
        <Password xsi:type="xsd:string">0JDOE0</Password>
      </ns1:getMessage>
   </soapenv:Body>
</soapenv:Envelope>
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

SOAP elements

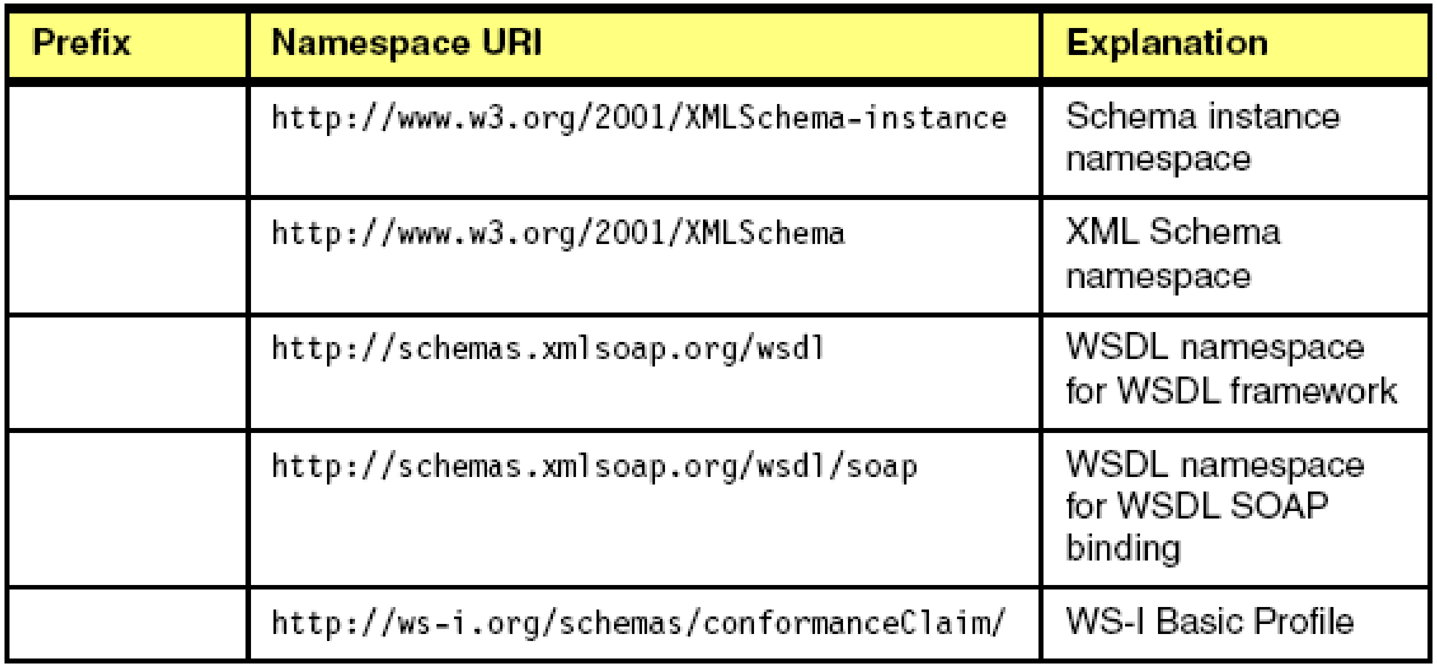
Namespaces

The use of namespaces plays an important role in SOAP message, because a message can include several different XML elements that must be identified by a unique namespace to avoid name collision.



Prefix	Namespace URI	Explanation
SOAP-ENV	`http://schemas.xmlsoap.org/soap/envelope/`	SOAP 1.1 envelope namespace
SOAP-ENC	`http://schemas.xmlsoap.org/soap/encoding/`	SOAP 1.1 encoding namespace

Namespaces



Prefix	Namespace URI	Explanation
	`http://www.w3.org/2001/XMLSchema-instance`	Schema instance namespace
	`http://www.w3.org/2001/XMLSchema`	XML Schema namespace
	`http://schemas.xmlsoap.org/wsdl`	WSDL namespace for WSDL framework
	`http://schemas.xmlsoap.org/wsdl/soap`	WSDL namespace for WSDL SOAP binding
	`http://ws-i.org/schemas/conformanceClaim/`	WS-I Basic Profile

URN

A unified resource name (URN) uniquely identifies the service to clients. It must be unique among all services deployed in a single SOAP server, which is identified by a certain network address. A URN is encoded as a universal resource identifier (URI). We commonly use the format urn:UniqueServiceID.urn:NextMessage is the URN of our message exchange Web service.

SOAP envelope

The envelope is the top element of the XML document representing the message with the following structure:

```
<SOAP-ENV:Envelope .... >
<SOAP-ENV:Header name="nmtoken">
<SOAP-ENV:HeaderEntry.... />
</SOAP-ENV:Header>
<SOAP-ENV:Body name="nmtoken">
[message payload]
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- Headers are a generic and flexible mechanism for extending a SOAP message in a decentralized and modular way without prior agreement between the parties involved. They allow control information to pass to the receiving SOAP server and also provide extensibility for message structures.
- Headers are optional elements in the envelope. If present, the element must be the first immediate child element of a SOAP envelope element. All immediate child elements of the header element are called header entries.
- There is a predefined header attribute called SOAP-ENV:mustUnderstand. The value of the mustUnderstand attribute is either 1 or 0. The absence of the SOAP mustUnderstand attribute is semantically equivalent to the value 0.

SOAP-BODY

- The SOAP body element provides a mechanism for exchanging information intended for the ultimate recipient of the message. The body element is encoded as an immediate child element of the SOAP envelope element. If a header element is present, then the body element must immediately follow the header element. Otherwise it must be the first immediate child element of the envelope element.
- All immediate child elements of the body element are called body entries, In the most simple case, the body of a basic SOAP message consists of:
- A message name.
- A reference to a service instance. In Apache SOAP, a service instance is identified by its URN. This reference is encoded as the namespace attribute.
- One or more parameters carrying values and optional type references.

Error handling

SOAP itself predefines one body element, which is the fault element used for reporting errors. If present, the fault element must appear as a body entry and must not appear more than once within a body element.

The fields of the fault element

- fault code is a code that indicates the type of the fault.
- fault string is a human-readable description of the fault.
- fault actor is an optional field that indicates the URI of the source of the fault.
- detail is an application-specific field that contains detailed information about the fault.

Data model

The SOAP Data model consists of

Simple XSD types

int, String, Date.

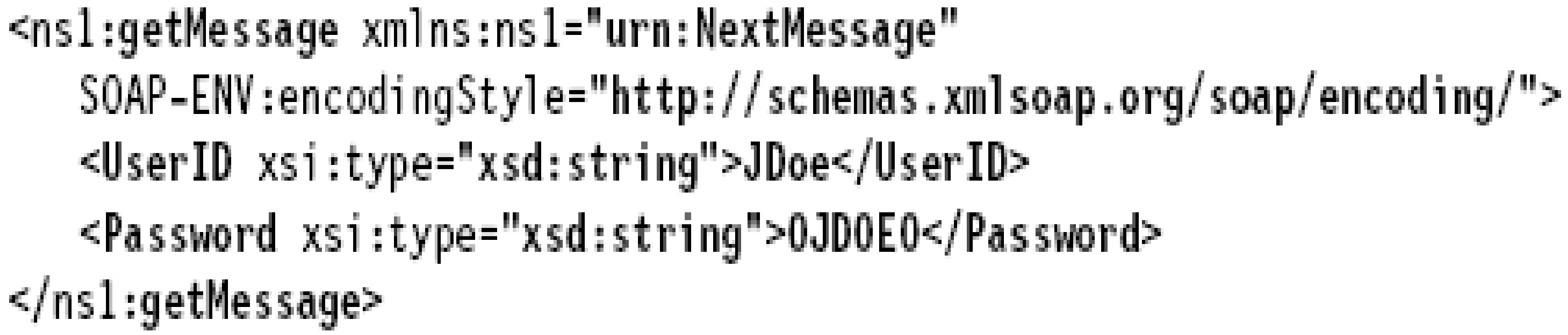
Compound types

Structs

Arrays

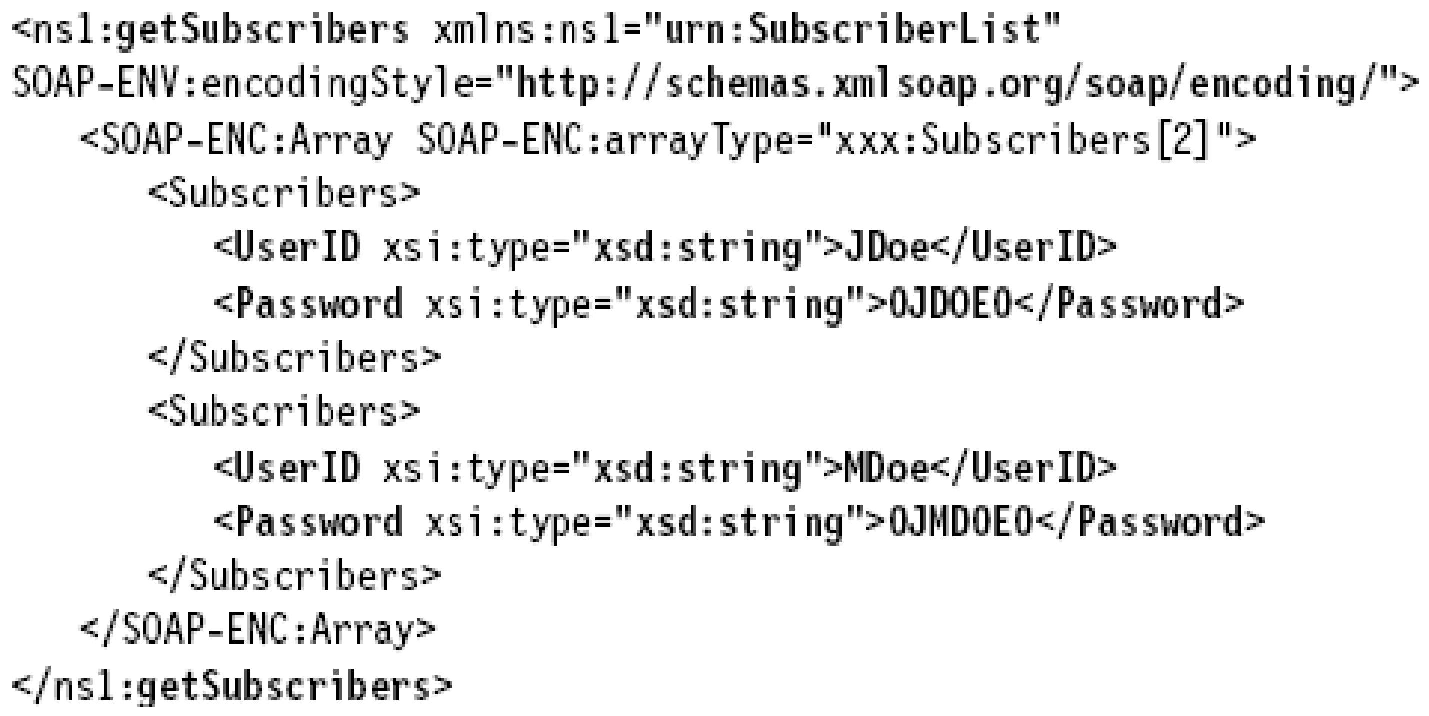
*All elements and identifiers comprising the SOAP data model are defined in the namespace SOAP-ENC***.**

SOAP Request Message



```
<ns1:getMessage xmlns:ns1="urn:NextMessage"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <UserID xsi:type="xsd:string">JDoe</UserID>
   <Password xsi:type="xsd:string">0JDOE0</Password>
</ns1:getMessage>
```

A SOAP request message with an array of structs



```
<ns1:getSubscribers xmlns:ns1="urn:SubscriberList"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENC:Array SOAP-ENC:arrayType="xxx:Subscribers[2]">
      <Subscribers>
         <UserID xsi:type="xsd:string">JDoe</UserID>
         <Password xsi:type="xsd:string">0JDOE0</Password>
      </Subscribers>
      <Subscribers>
         <UserID xsi:type="xsd:string">MDoe</UserID>
         <Password xsi:type="xsd:string">0JMDOE0</Password>
      </Subscribers>
   </SOAP-ENC:Array>
</ns1:getSubscribers>
```

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Communication styles

Document ***Also known as message-oriented style:***

This style provides a lower layer of abstraction, and requires more programming work. The in parameter is any XML document; the response can be anything (or nothing). This is a very flexible communication style that provides the best interoperability.

RPC

The remote procedure call is a synchronous invocation of operation returning a result, conceptually similar to other RPCs.

Encodings

In distributed computing environments, encodings define how data values defined in the application can be translated to and from a protocol format. We refer to these translation steps as serialization and deserialization, or, synonymously,marshalling and unmarshalling

SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in a specific programming language into SOAP XML and vice versa.

Types Of Encoding

SOAP encoding

The SOAP encoding enables marshalling/unmarshalling of values of data types from the SOAP data model.

Literal

The literal encoding is a simple XML message that does not carry encoding information. Usually, an XML Schema describes the format and data types of the XML message.

Literal XML

The literal XML encoding enables direct conversion of existing XML DOM tree elements into SOAP message content and vice versa

XMI

XML metadata interchange (XMI) is defined by the Apache SOAP implementation.

Messaging modes

The two styles (RPC, document) and two most common encodings (encoded,literal) can be freely intermixed to what is called a SOAP messaging mode

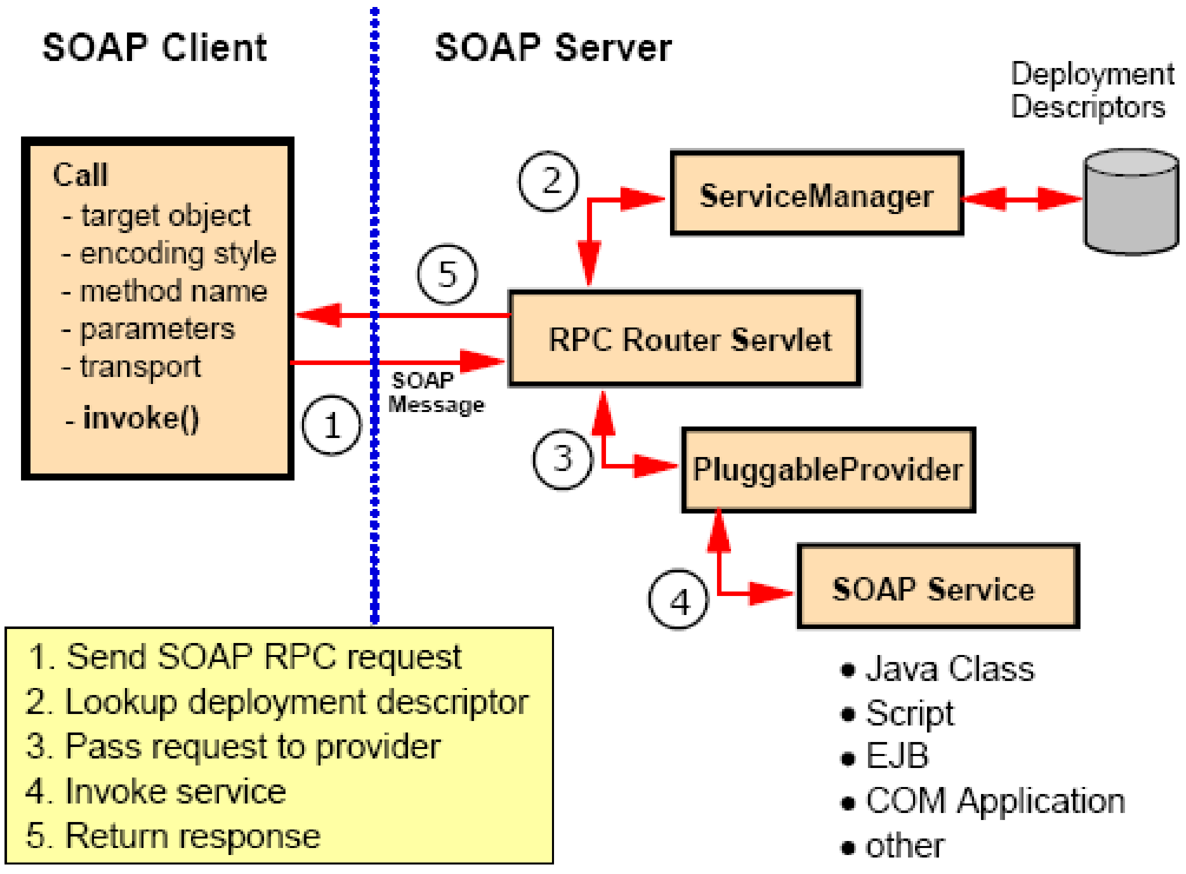
Document/literal—Provides the best interoperability between Java and non-Java implementations, and is also recommended for Java-to-Java applications.

RPC/literal—Possible choice between Java implementations. Although RPC/literal is WS-I compliant, it is not frequently used in practice

RPC/encoded—Early Java implementations (Web Sphere Application Server Versions 4 and 5.0) supported this combination, but it does not provide interoperability with non-Java implementations.

Document/encoded—Not used in practice.

SOAP client and server interaction



1. Send SOAP RPC request
2. Lookup deployment descriptor
3. Pass request to provider
4. Invoke service
5. Return response

Web service invocation using RPC involves the following steps

1. A SOAP client generates a SOAP RPC request document and sends it to a RPC router.

2. The router contacts the service manager to obtain a deployment descriptor.

3. Based on routing information from the deployment descriptor, the router forwards the request to a service provider.

4. The service provider invokes the requested service and returns a result to the router.

5. The router sends the service result message to the client

WSDL -Overview

WSDL enables a service provider to specify the following characteristics of a Web service:

1. Name of the Web service and addressing information
2. Protocol and encoding style to be used when accessing the public operations of the Web service
3. Type information: Operations, parameters, and data types comprising the interface of the Web service, plus a name for this interface

WSDL document

Types

A container for data type definitions using some type system, such as XML Schema.

Message

An abstract, typed definition of the data being communicated. A message can have one or more typed parts.

Port type

An abstract set of one or more operations supported by one or more ports.

Operation

An abstract description of an action supported by the service that defines the input and output message and optional fault message.

WSDL document

Binding

A concrete protocol and data format specification for a particular port type. The binding information contains the protocol name, the invocation style, a service ID, and the encoding for each operation.

Service

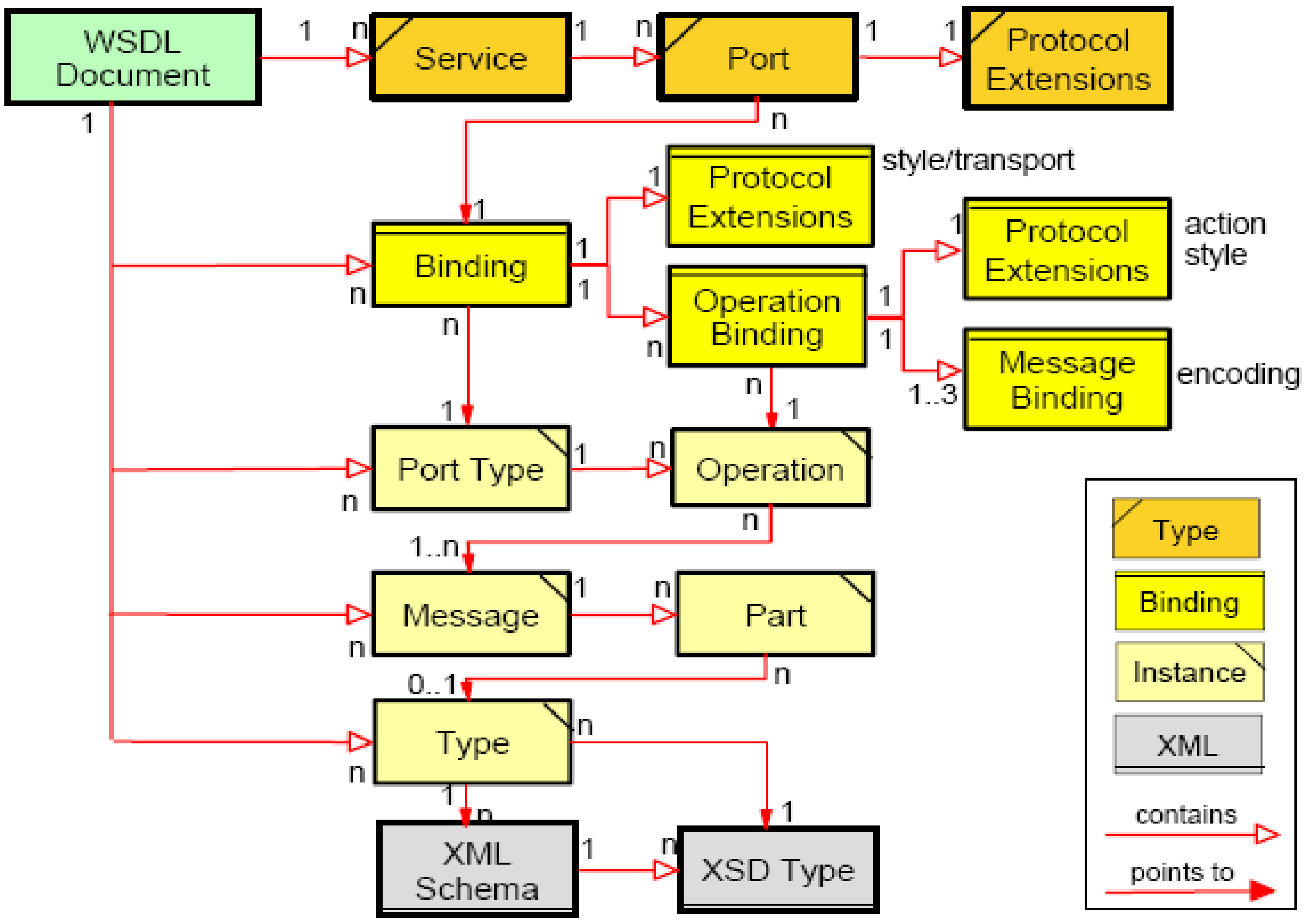
A collection of related ports.

Port

A single endpoint, which is defined as an aggregation of a binding and a network address.

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

WSDL document anatomy



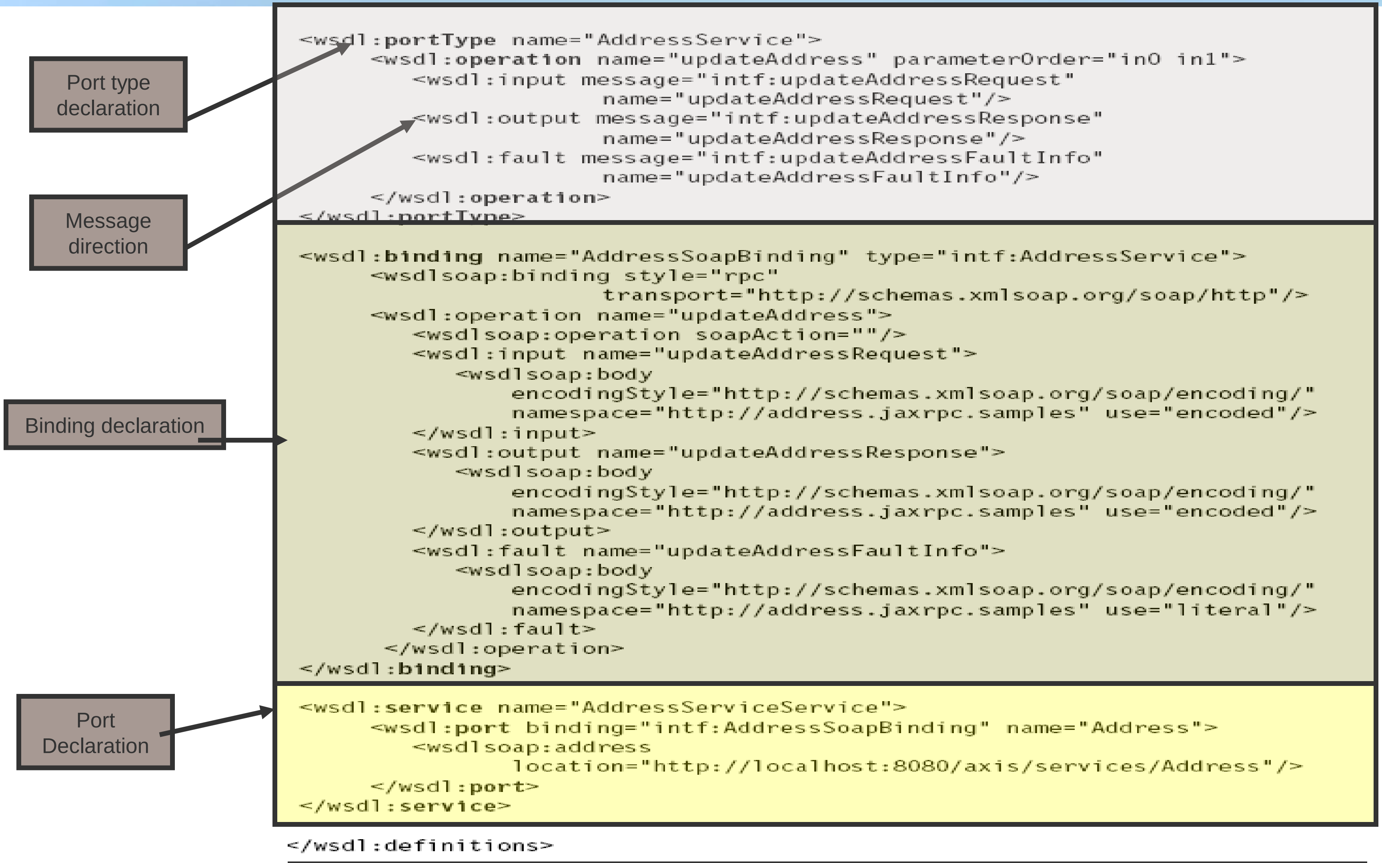
WSDL definition

Root node
always
Definitions

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://address.jaxrpc.samples"
   xmlns:apachesoap="http://xml.apache.org/xml-soap"
   xmlns:impl="http://address.jaxrpc.samples"
   xmlns:intf="http://address.jaxrpc.samples"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <wsdl:types>
   <schema targetNamespace="http://address.jaxrpc.samples"
          xmlns="http://www.w3.org/2001/XMLSchema">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType name="AddressBean">
    <sequence>
     <element name="street" nillable="true" type="xsd:string"/>
     <element name="zipcode" type="xsd:int"/>
    </sequence>
   </complexType>
   <element name="AddressBean" nillable="true" type="impl:AddressBean"/>
  </schema>
  <import namespace="http://www.w3.org/2001/XMLSchema"/>
 </wsdl:types>

 <wsdl:message name="updateAddressRequest">
      <wsdl:part name="in0" type="intf:AddressBean"/>
      <wsdl:part name="in1" type="xsd:int"/>
 </wsdl:message>
 <wsdl:message name="updateAddressResponse">
      <wsdl:part name="return" type="xsd:string"/>
 </wsdl:message>
 <wsdl:message name="updateAddressFaultInfo">
      <wsdl:part name="fault" type="xsd:string"/>"
 </wsdl:message>
```

Message
declaration

WSDL definition



Port type declaration
Message direction
Binding declaration
Port Declaration

```
<wsdl:portType name="AddressService">
     <wsdl:operation name="updateAddress" parameterOrder="in0 in1">
        <wsdl:input message="intf:updateAddressRequest"
                     name="updateAddressRequest"/>
        <wsdl:output message="intf:updateAddressResponse"
                     name="updateAddressResponse"/>
        <wsdl:fault message="intf:updateAddressFaultInfo"
                     name="updateAddressFaultInfo"/>
     </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="AddressSoapBinding" type="intf:AddressService">
     <wsdlsoap:binding style="rpc"
                     transport="http://schemas.xmlsoap.org/soap/http"/>
     <wsdl:operation name="updateAddress">
        <wsdlsoap:operation soapAction=""/>
        <wsdl:input name="updateAddressRequest">
           <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://address.jaxrpc.samples" use="encoded"/>
        </wsdl:input>
        <wsdl:output name="updateAddressResponse">
           <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://address.jaxrpc.samples" use="encoded"/>
        </wsdl:output>
        <wsdl:fault name="updateAddressFaultInfo">
           <wsdlsoap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="http://address.jaxrpc.samples" use="literal"/>
        </wsdl:fault>
      </wsdl:operation>
</wsdl:binding>

<wsdl:service name="AddressServiceService">
     <wsdl:port binding="intf:AddressSoapBinding" name="Address">
        <wsdlsoap:address
               location="http://localhost:8080/axis/services/Address"/>
     </wsdl:port>
</wsdl:service>

</wsdl:definitions>
```

Web services for J2EE overview

WSEE leverages J2EE technologies defining the needed mechanism to standardize a deployment model for Web services. This standardization wants to achieve the interoperability across different compliant J2EE platforms

Although WSEE does not restrict any implementation, it only defines two:

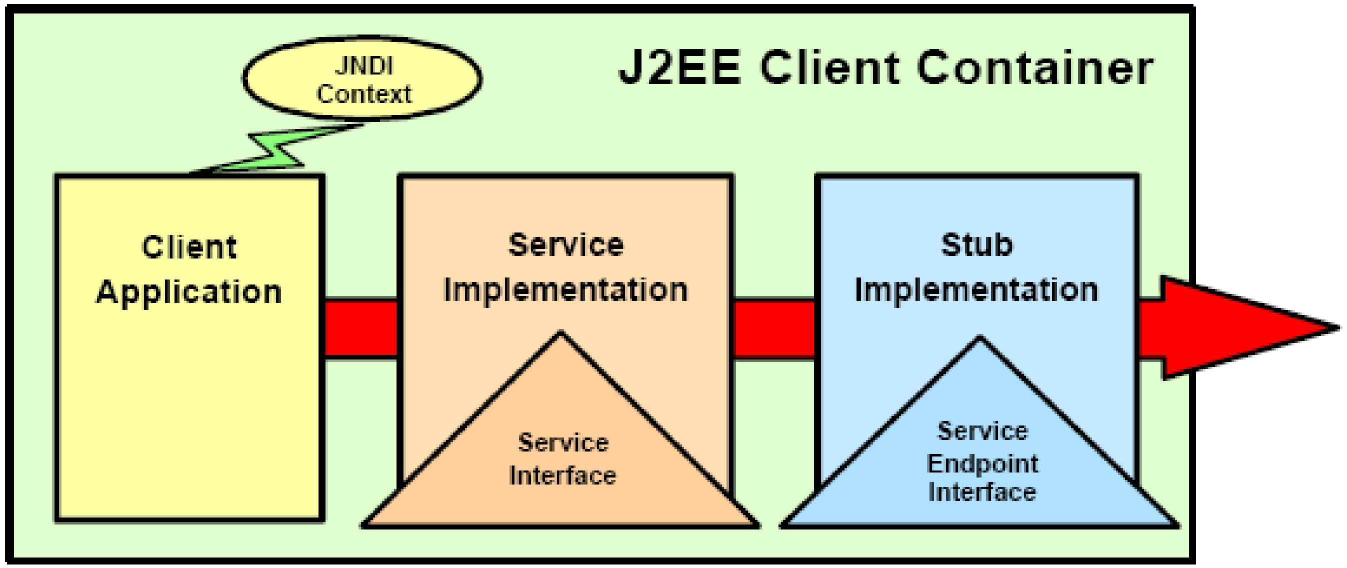
- Stateless session EJB in an EJB container
- Java class running in a Web container

Web services for J2EE overview

For these two implementation models, and for both the client and server sides, the specification details are:

- How J2EE server components can be defined to be accessible as Web services
- How J2EE client components and applications can use JAX-RPC to access Web services
- The assembly of the components with Web services
- How J2EE server components can be described as Web services
- How J2EE client components can be described for accessing Web
- services using JAX-RPC

J2EE Web service client components.



Client concepts

- The J2EE client container provides the WSEE runtime that is used by a client to access and invoke Web service methods. The client uses a JNDI lookup to find a service object.
- A service object implements the service interface as defined by JAX-RPC. The client gets a stub or a proxy by using the factory function of the service object.
- A stub represents the Web service instance on the client side. Regardless, the Web service client should use the Web service interface and not the stub. Web service stubs are generated during deployment and are specific to the client runtime.

Client concepts

WSEE specifies three mechanisms for invoking a Web service:

Static stub

Static, because the stub is created before runtime. This requires complete WSDL knowledge.

Dynamic proxy

Dynamic, because the proxy class is created during runtime. Only a partial WSDL definition is required (port type and bindings).

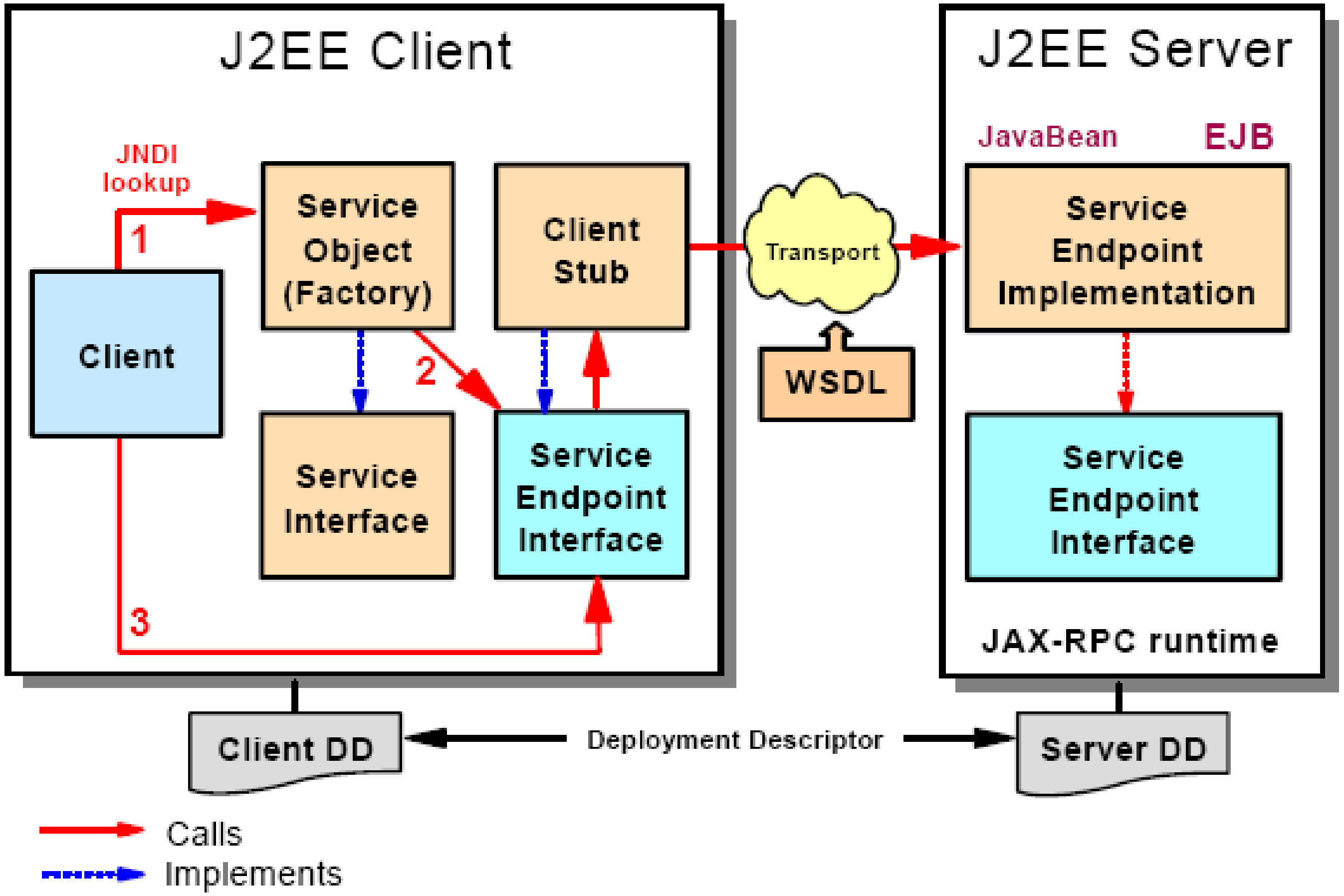
Dynamic invocation interface

Does not require WSDL knowledge. The signature of the remote procedure or the name of the service are unknown until runtime.

Static stub

- The static stub client is statically bound to a service endpoint interface (SEI), a WSDL port, and a port component. The stub is used by the client to invoke the Web service. A static client is also tied to a specific protocol and transport type.
- The static stub is the easiest of the three styles in respect to development. But even a small change in the WSDL document provokes the client useless; the stub must be regenerated.

Static client calling sequence



Static client calling sequence

- The client makes an JNDI lookup to get an instance of the service object, which implements a service interface.
- The client uses a factory method of the service object to retrieve the client stub. The client stub implements the SEI.
- The client invokes the Web service through the SEI.

Dynamic proxy

- A dynamic proxy is not tied to an specific stub and, therefore, does not have the restrictions of a static stub client.
- The dynamic proxy client stub is generated at runtime when a Web service method is called.
- The Web service WSDL file must be available to the JAX-RPC runtime, because the WSDL file is parsed to create the specific stub.
- After the dynamic proxy has been generated by the runtime, the client uses the same mechanism to invoke the Web service as with the static stub implementation.

Dynamic invocation interface (DII)

- The dynamic invocation interface uses a javax.xml.rpc.Call instance for dynamic invocation. Unlike the two previous clients, the DII has to previously configure the Call object
- Operation name
- Parameters names, types, and modes
- Port type and address of a target service endpoint
- Protocol and transport properties

Packaging

WSEE defines the artifacts for the Web services client packaging. The packaging structure is specific to the used Web services client type. A WSEE client deployment package must contain

- WSDL file
- Service endpoint interface class
- Service implementation bean class, and dependent classes
- Web services client deployment descriptor
- JAX-RPC mapping file

Web service for J2EE client deployment descriptor

The WSEE specification does not define the client deployment descriptor name. Rational Application Developer Version 6 (also Rational Web Developer) adds the WSEE deployment information for J2EE 1.4 Web service clients to existing client project deployment descriptors. Those deployment descriptors are used as defined in the J2EE specification:

- Web service EJB client—META-INF/ejb-jar.xml
- Web service Web client—WEB-INF/web.xml

For J2EE 1.3 client projects, a separate deployment descriptor is generated:

- Web service EJB client—META-INF/webservicesclient.xm
- Web service Web client—WEB-INF/webservicesclient.xm

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

WSEE client deployment descriptor

WSEE client deployment descriptors contain service reference entries. The J2EE client container will use these definitions to deploy the Web services client.

- Description—Web service description.
- Service reference—The logical name of the reference used by JNDI to look up the service reference.
- Service type—The fully qualified name of the service interface; returned by the JNDI lookup.
- WSDL definition—The location of the service WSDL.
- JAX-RPC mapping—The location of the JAX-RPC mapping file.
- Service port—The qualified name of the referenced WSDL service.
- Ports—The port component provides the local representation of our Web service, the service endpoint interface.

JAX-RPC mapping deployment descriptor

The name for the JAX-RPC mapping deployment descriptor is not specified by WSEE. Application Developer uses these names for the mapping file:

- Web service EJB client: META-INF/<WSDL-Filename>_mapping.xml
- Web service Web client: WEB-INF/<WSDL-Filename>_mapping.xml

JAX-RPC mapping deployment descriptor

Package mapping

Contains the relation between the XML namespace and Java packages.

Type mapping

Contains the type mapping between Java types and XML types.

Service interface mapping

Contains the mapping for the WSDL service definition. The service endpoint interface mapping

JAX-RPC mapping deployment descriptor

- The package mapping—Contains the relation between the XML namespace and Java packages.
- The type mapping—Contains the type mapping between Java types and XML types.
- The service interface mapping—Contains the mapping for the WSDL service definition.
- The service endpoint interface mapping

Server programming model

- The server programming model provides the server guidelines for standardizing the deployment of Web services in a J2EE server environment.
- Web container—A Java class according to a JAX-RPC servlet-   based service
- EJB container—A stateless session EJB

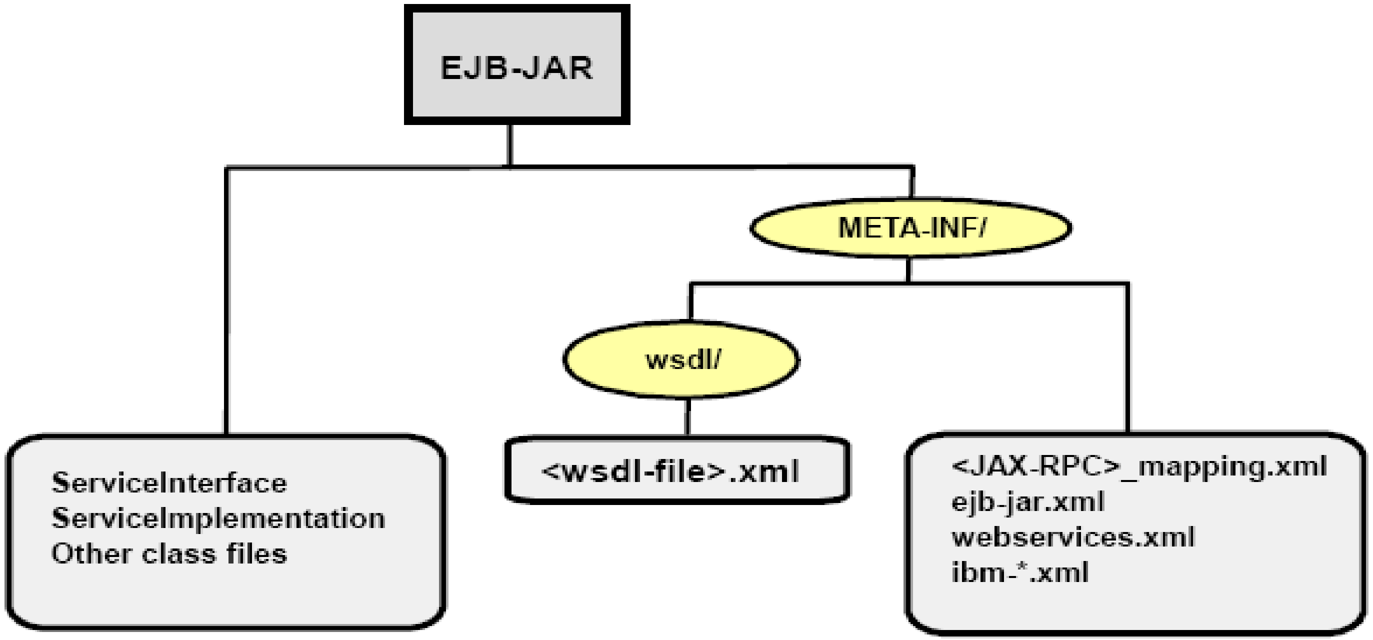
Server concepts

- WSDL definition—Provides the description of a Web service.
- Service endpoint interface (SEI)—Defines the operations available for a Web service. Must follow JAX-RPC mapping rules.
- Service implementation bean—Implements the SEI methods to provide the business logic of a Web service.
- Security role references—Provides instance-level security check across the different modules.

EJB container programming model

- The stateless session EJB must have a default public constructor, a default EJB create method, and one or more EJB remote methods.
- The service endpoint interface must be a subset of the methods of the remote interface of the session EJB. The remote interface does not have to implement the endpoint interface. The methods must be public, but neither final nor static.
- The stateless session EJB must be a stateless object.
- The class must be public, but neither final nor abstract.
- The class must not have a finalize method.

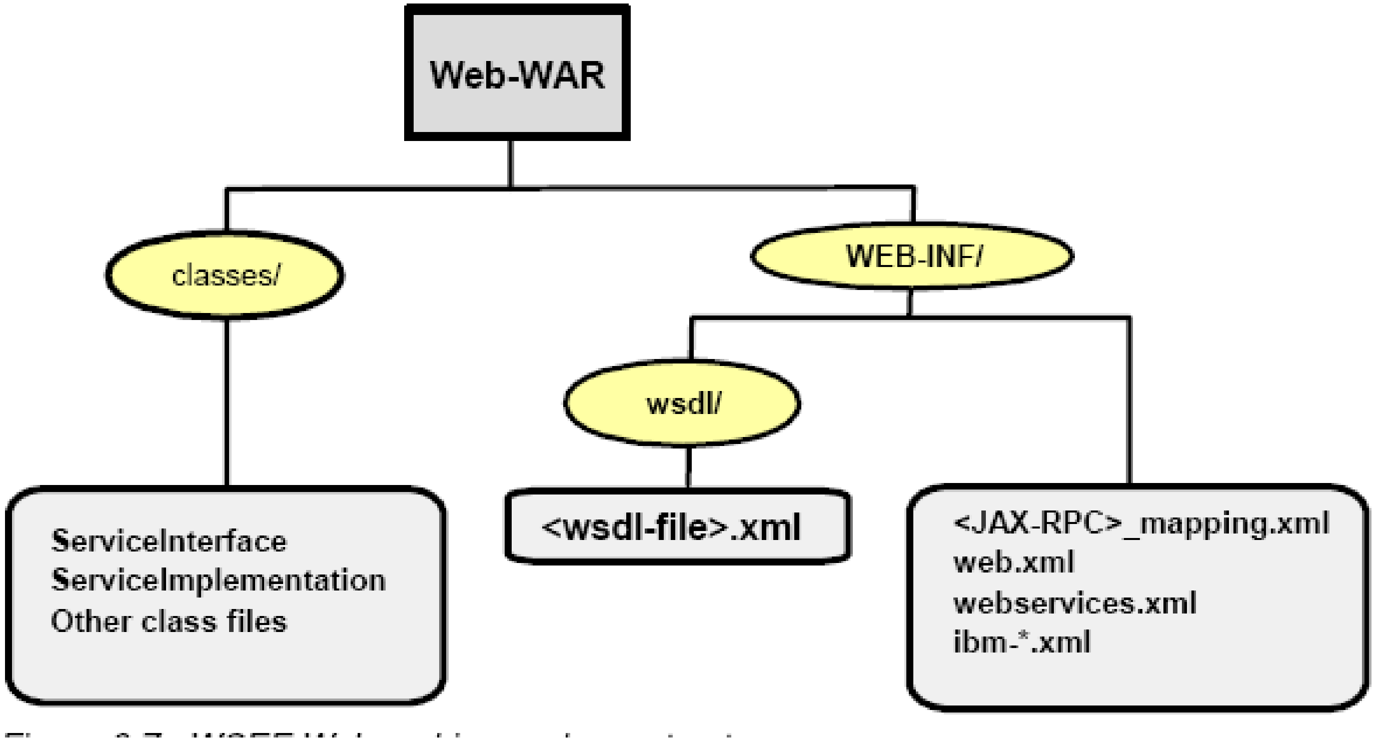
WSEE EJB archive package structure



Web container programming model

- The Java class must have a default public constructor.
- The Java class must implement the method signatures of the service endpoint interface. Only those methods are exposed to the client.
- The Java class must be stateless.
- The Java class must be public, but neither final nor abstract.
- The Java class must not have a finalize method.

WSEE Web archive package structure



Packaging

- Service endpoint interface classes
- Generated service interface, and dependent classes
- WSDL file
- Web services client deployment descriptor
- JAX-RPC mapping file

Web service deployment descriptor

Service name

The unique name of the service. This name is the same name of the wsdl:sevice element.

JAX-RPC mapping

The location of the WSDL-Java mapping file.

WSDL file

The location of the WSDL description of the service.

Port

The port component defines the server view, providing the access point and implementation details.

name

The unique name of the WSDL port element for this service. This name is the same name of the wsdl:port element.

Web service deployment descriptor

qname

The qualified name of the referenced WSDL port.

SEI

The fully qualified class name of the service endpoint interface.

bean class

The implementation name of the Web service. This name must match the name of the ejb-name element stated in the ejb-jar.xml file. For the Web container programming model, the servlet-link element is used in the webservices.xml and the name must match the name of the servlet-name stated in the web.xml file.

Handlers

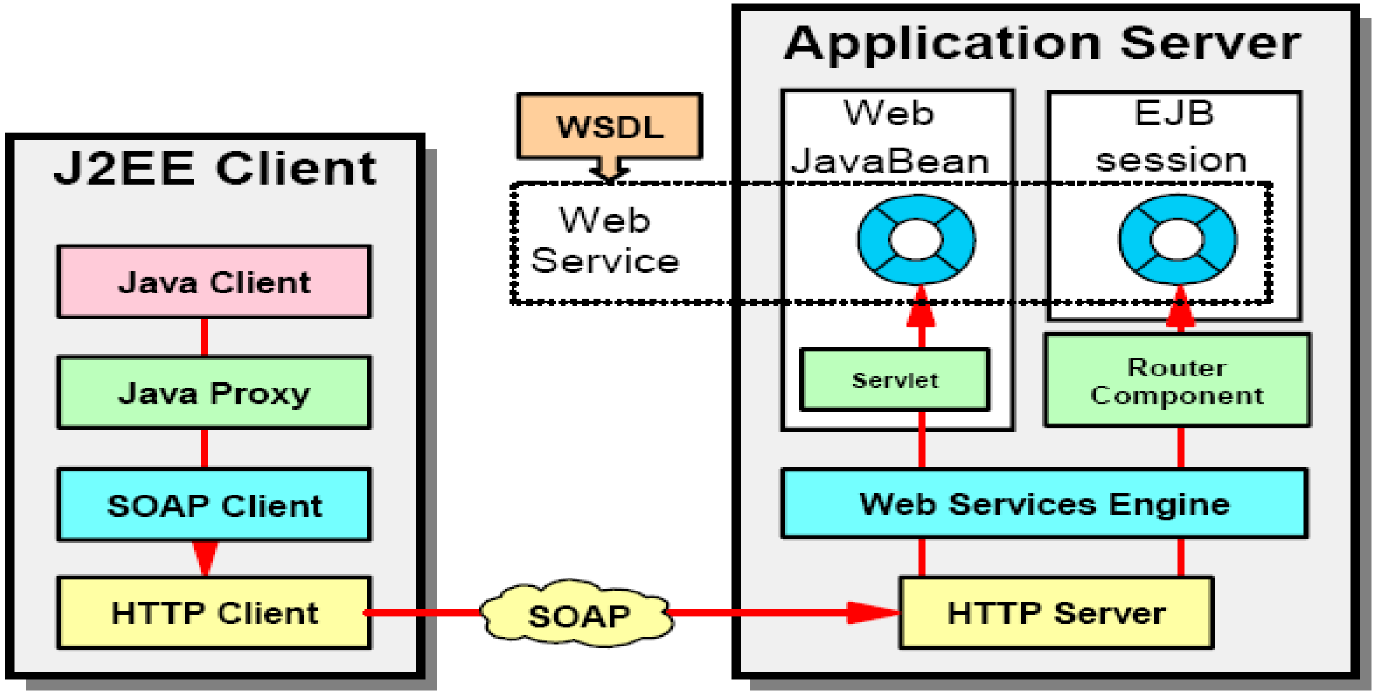
The handlers associated with the Web service reference.

WSEE implementations in WebSphere

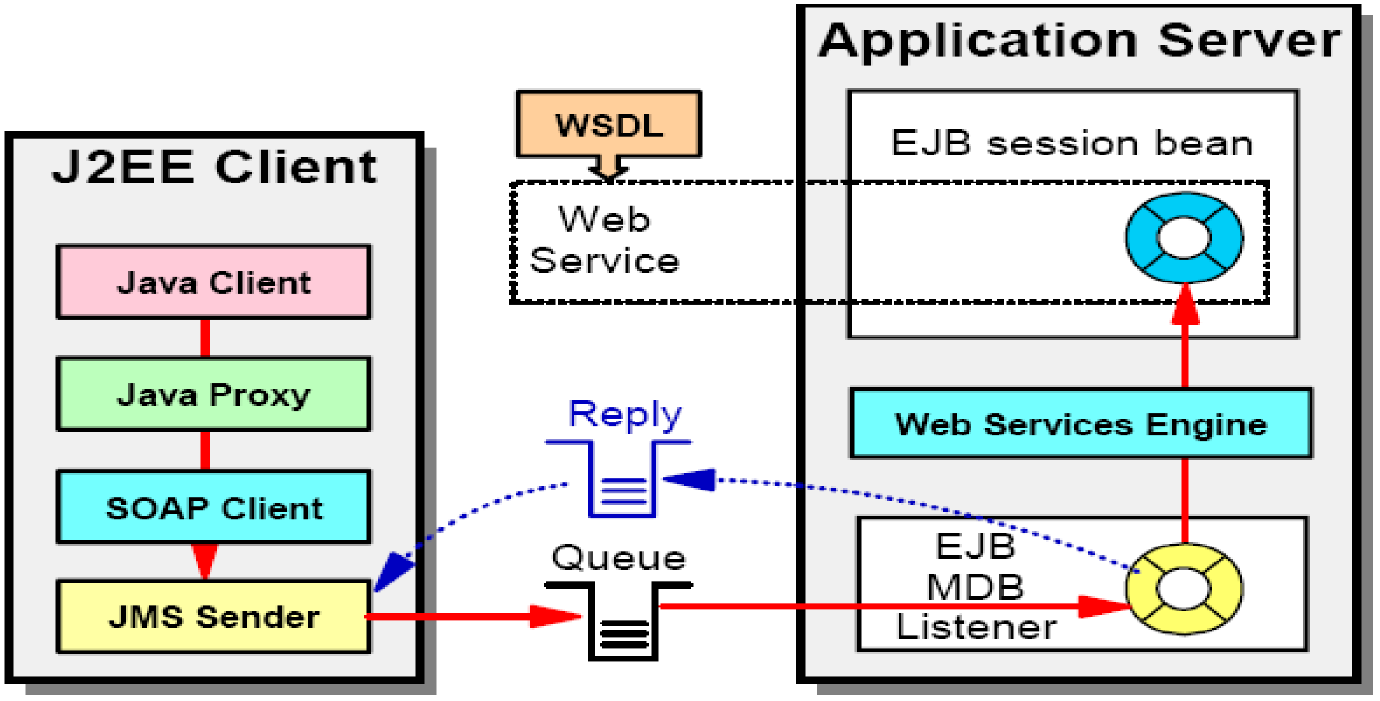
WebSphere Application Server 6.0 fully supports the Web Services for J2EE 1.1 specification.

- SOAP over HTTP
- SOAP over JMS are supported.

SOAP over HTTP



SOAP over JMS



UDDI

- UDDI stands for Universal Description, Discovery, and Integration, and is the name for a specification that defines a way to store and retrieve information about a business and its technical interfaces, in our case, Web services.
- UDDI is based on existing standards, such as XML and SOAP. It is a technical discovery layer. It defines:
    - The API that can be used to access registries with this structure
    - The organization and project defining this registry structure and its API
    - The structure for a registry of service providers and services

Static versus dynamic Web services

Static

- Static Web services mean that the service provider and the service requestor know about each other at design time. There is a WSDL document that was found in a UDDI registry, or, more often, directly sent to the client application developer by the provider for further use in the development tool. During runtime, it is very clear (mostly hard coded) what the URL (access point) of the partner is.

Dynamic

- Dynamic Web services describe the fact that at design and development time the client does not know the explicit server and business entity where it will invoke a service. The client only knows an interface to call and finds one or more concrete providers for that kind of service through exploring UDDI registries at runtime.

UDDI registry structure

The data to be stored can be divided into six types of information that build up the data model of the registry:

Business entity

- The list of business entities is similar to the white and yellow pages. A business entity describes a company or organization.

Business service

- This is non-technical information about a service that is provided.

Binding template

- This contains service access information. These are the technical Web service descriptions relevant for application developers who want to find and invoke a Web service.

tModel

- A *tModel* (technical model) is a technical fingerprint holding metadata about type specifications and categorization information.

Taxonomy

A taxonomy is a scheme for categorization. There is a set of standard taxonomies, such as the North American Industry Classification System (NAICS) and the Universal Standard Products and Services Classification (UNSPSC).

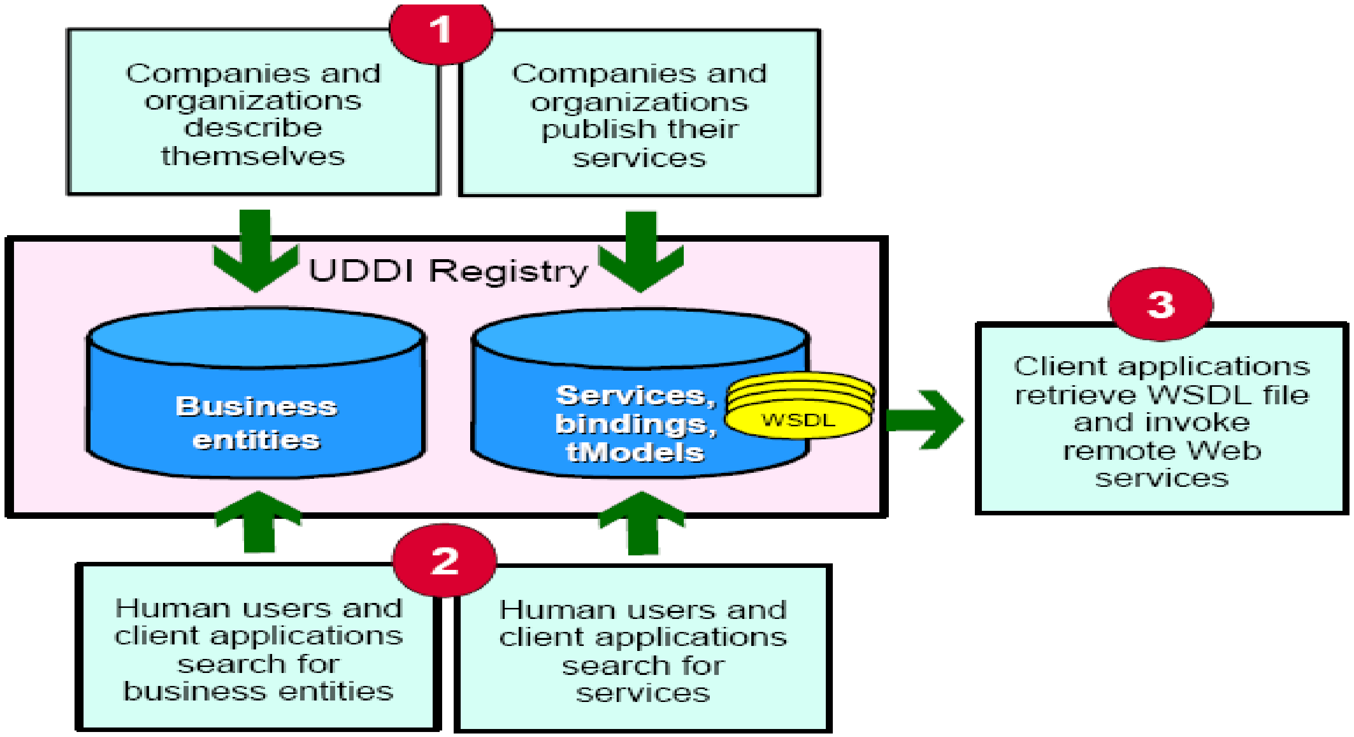
Publisher assertions

These are also called business relationships. There are different kinds of relationships: Parent-child, peer-peer, and identity.

Interactions with UDDI

- Publishing information
- Finding information
- Using the obtained information

Interactions with UDDI registries


1
Companies and organizations describe themselves
Companies and organizations publish their services
UDDI Registry
Business entities
Services, bindings, tModels
WSDL
3
Client applications retrieve WSDL file and invoke remote Web services
2
Human users and client applications search for business entities
Human users and client applications search for services

Finding information

- Clients have a number of possible ways to explore a registry. Humans do that by using HTML clients of the registry, and applications use UDDI APIs to do that automatically, as described in “Java APIs for dynamic UDDI interactions”

Java APIs for dynamic UDDI interactions

- The IBM UDDI Version 3 Client for Java is the preferred API for accessing UDDI using Java.

- Another Java API that can be used to access UDDI repositories is UDDI4J. A number of companies, such as IBM, HP, and SAP, officially support the UDDI4J implementation

UDDI4J

- UDDI4J is a Java class library that provides support for an API that can be used to interact with a UDDI registry. This class library provides classes that can be used to generate and parse messages sent to and received from a UDDI server.
- The central class in this set of APIs   org.uddi4j.client.UDDIProxy. It is a proxy for the UDDI server that is accessed from client code.
- You can use this proxy to interact with the server and possibly find and download WSDL files.

Prerequisites

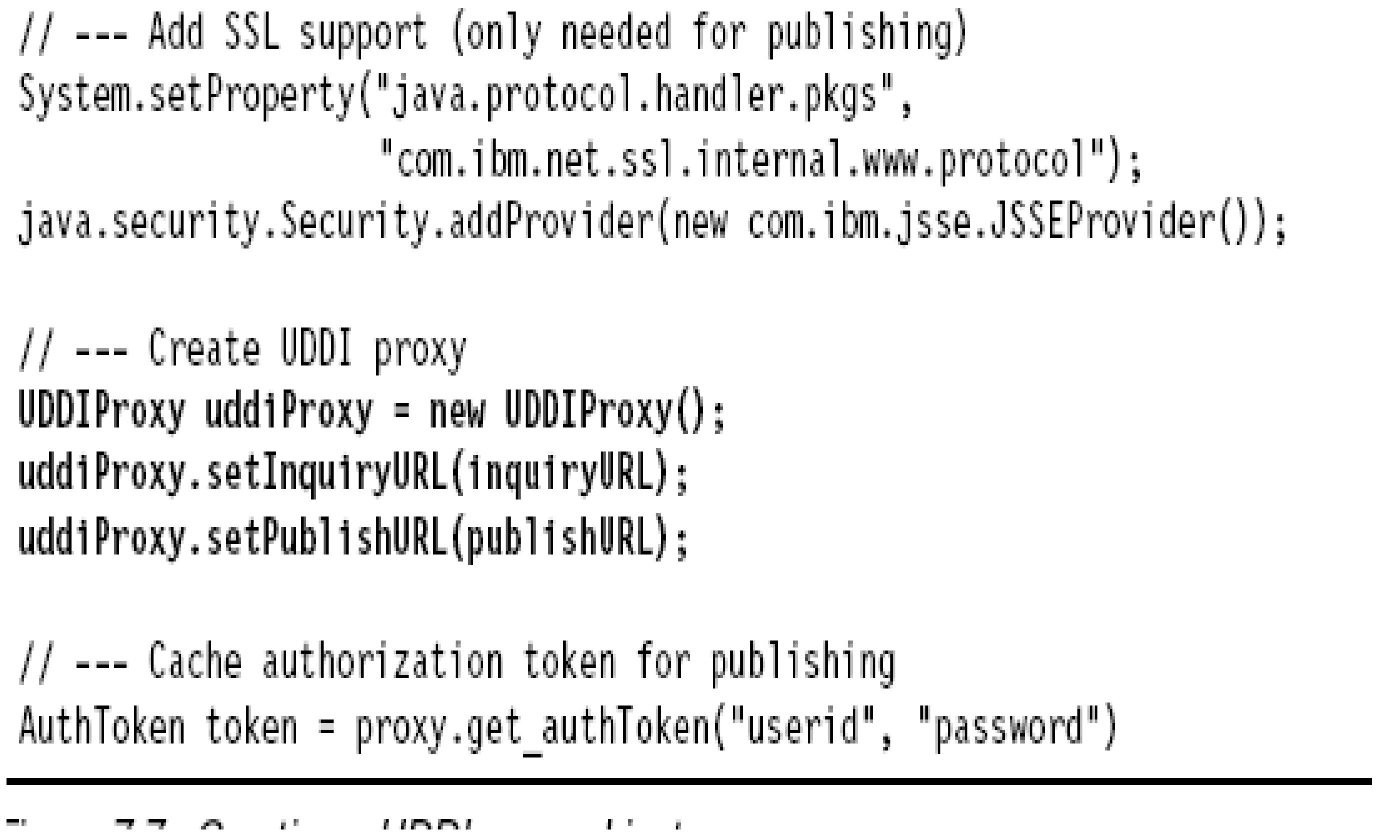
Apache Axis

Apache SOAP 2.3

HP SOAP

Writing UDDI clients

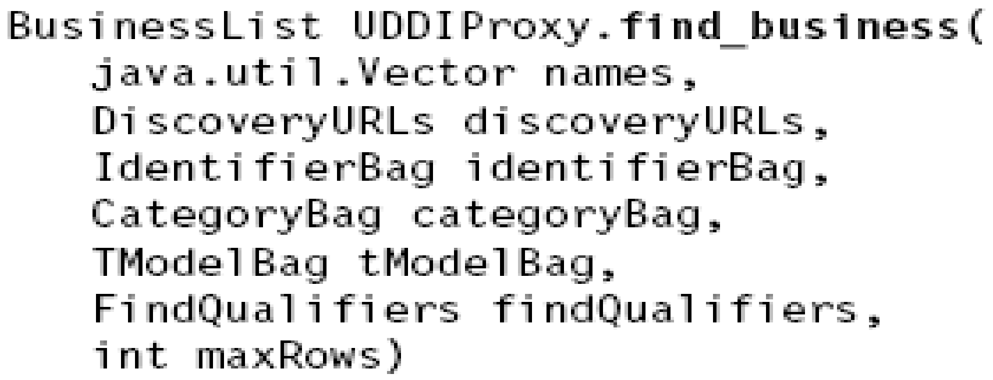
Connect to the server by creating a proxy object.



```
// --- Add SSL support (only needed for publishing)
System.setProperty("java.protocol.handler.pkgs",
                   "com.ibm.net.ssl.internal.www.protocol");
java.security.Security.addProvider(new com.ibm.jsse.JSSEProvider());

// --- Create UDDI proxy
UDDIProxy uddiProxy = new UDDIProxy();
uddiProxy.setInquiryURL(inquiryURL);
uddiProxy.setPublishURL(publishURL);

// --- Cache authorization token for publishing
AuthToken token = proxy.get_authToken("userid", "password")
```

Finding information

Finding business entities

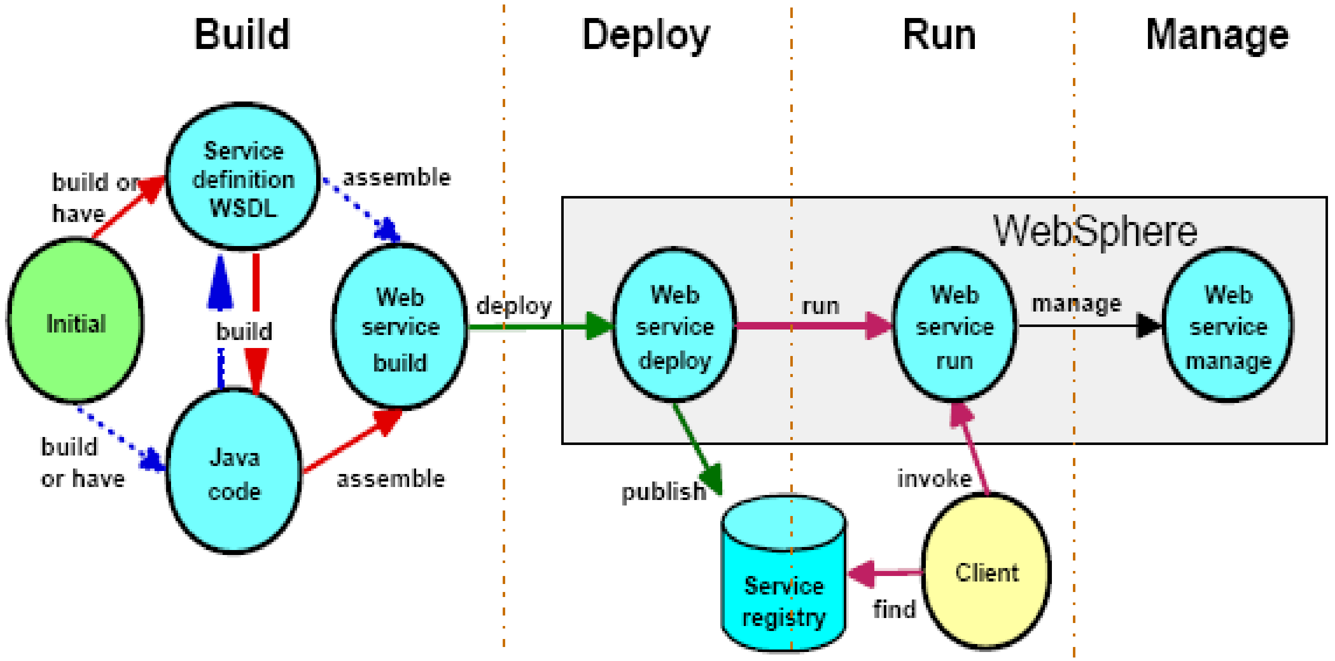


```
BusinessList UDDIProxy.find_business(
    java.util.Vector names,
    DiscoveryURLs discoveryURLs,
    IdentifierBag identifierBag,
    CategoryBag categoryBag,
    TModelBag tModelBag,
    FindQualifiers findQualifiers,
    int maxRows)
```

Development -4-Phases

- The build phase includes development and testing of the Web service application, including the definition and functionality of the service.
- The deploy phase includes publication of the service definition, the WSDL document, and deployment of the runtime code of the Web service.
- The run phase includes finding and invoking the Web service.
- The manage phase includes the management and administration of the Web service. This includes performance measurement and maintenance of the Web service.

Web services development



Build phase

The red (solid) path

From the initial state, we build or already have Java code. Using this Java code, we build the service definition (WSDL document) with the business methods that we want to expose. After we have generated the WSDL document, we assemble the Web service application. This approach is called bottom-up development.

The blue (dashed) path

From the initial state, we build or already have a service definition, a WSDL document. Using this WSDL document, we build or adapt the Java code to implement that service. After we have implemented the code, we assemble the Web service application. This approach is called top-down development.

Deploy phase

In this phase, we deploy the Web service to an application server. Deploying a Web service makes it accessible by clients.

However, these clients have to be aware of the newly installed Web service.

The next step in this phase is to publish the Web service. The publication can be done through a private or public UDDI registry, using a WSIL document, or by directly providing the information about the new service to consumers, for example, through e-mail.

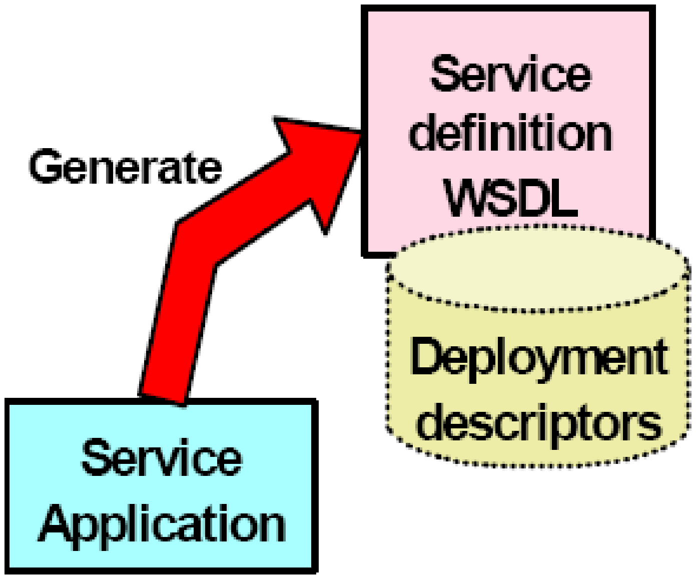
Run phase

- In this phase, the Web service is operative and is
- invoked by clients that require the functionality offered by this service.
- Manage phase
- The final phase is the management phase where we cover all the management and administration tasks of the Web service.

Bottom Up

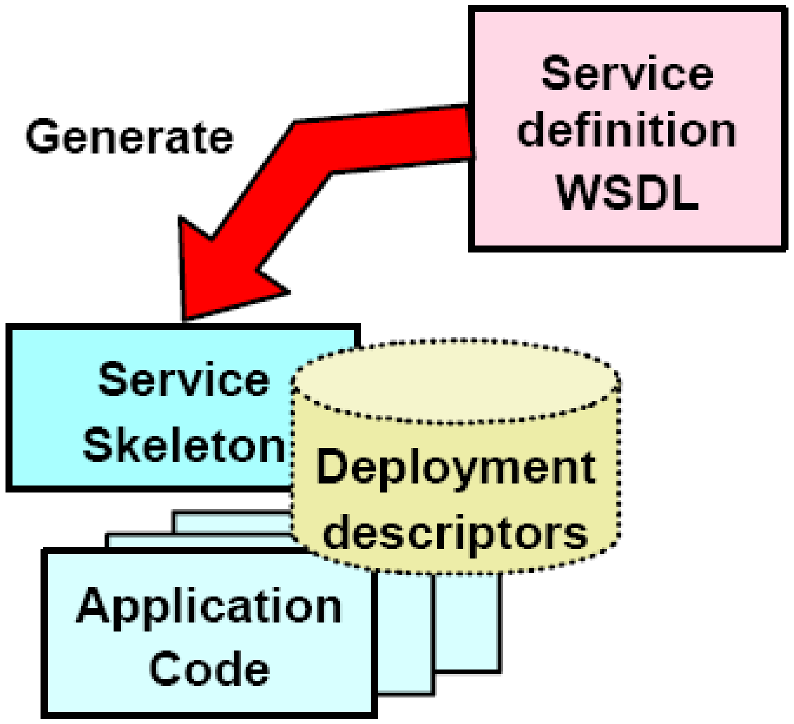
The bottom-up approach is the most common way to build a Web service.

We start with a business application that is already developed, tested, and running on the company systems



Top-down

The top-down approach is commonly used when we have a standard service definition and we want to implement this definition to provide the requested service.



Web service client

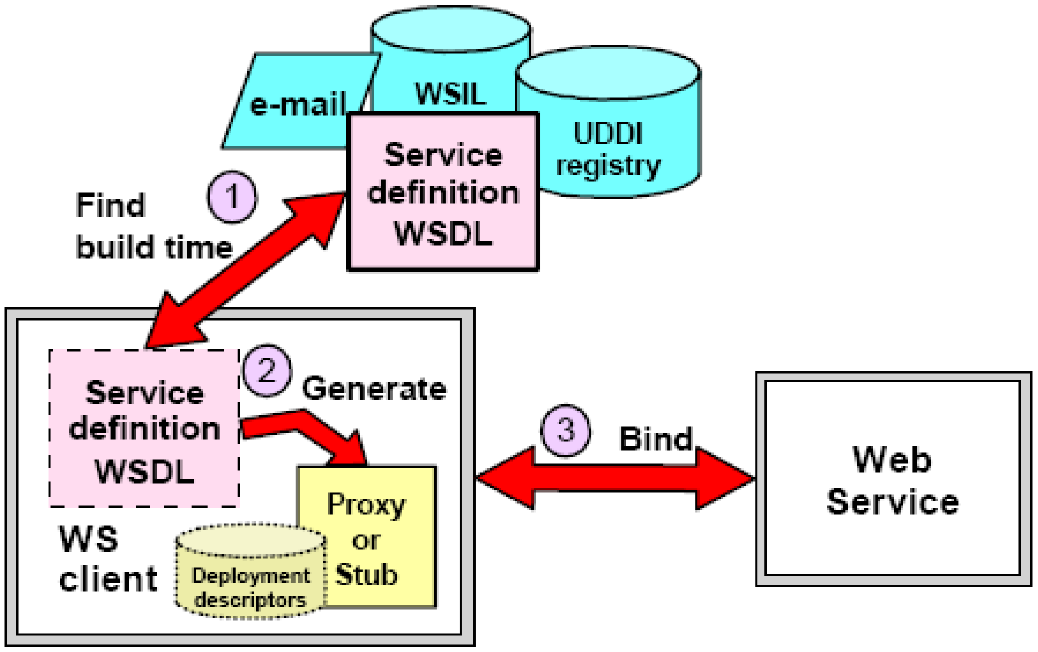
- Static client
- Dynamic client with known service type
- Dynamic client with unknown service type

Static client

- The static client has a static binding created at build time.
- This is made possible, because in the development phase, we know the interface, the binding method, and the service endpoint of the Web service that we are going to invoke.
- We also decide that we only use this client for that specific service and nothing else.
- Therefore, in these cases, the best solution is to use a static client.

The steps to build a static service client :

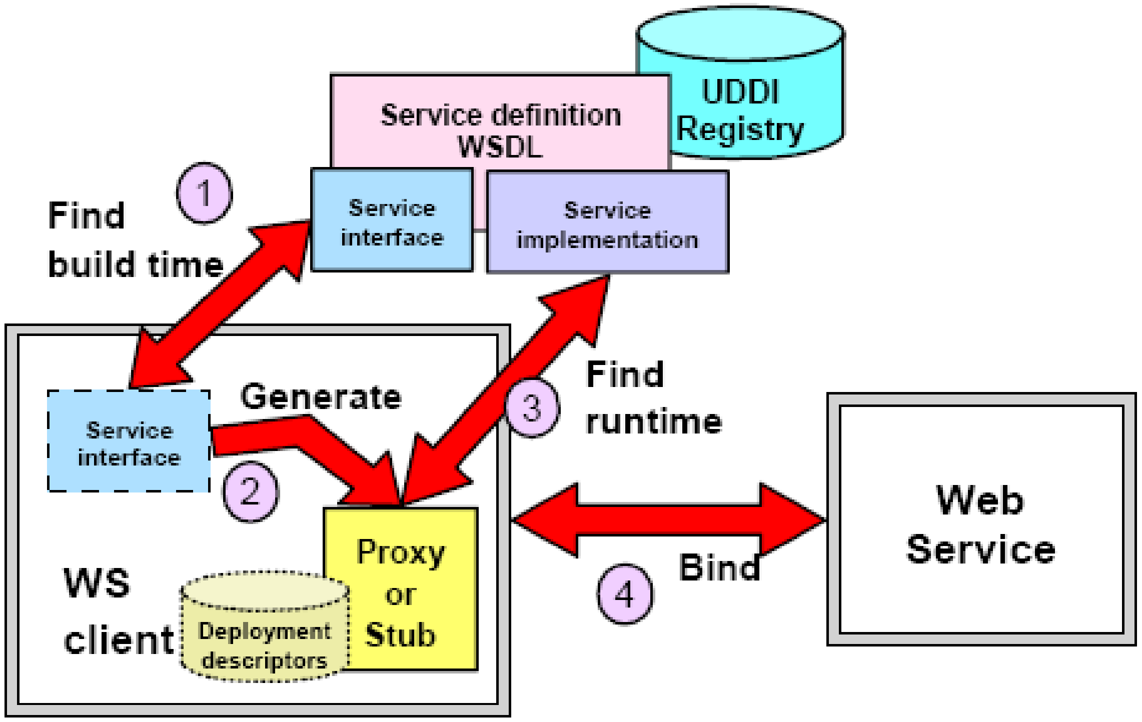
- Manually find the service definition or WSDL
- Generate the service proxy or stub
- Test the client



Dynamic client with known service type

- Manually find the service interface definition of the WSDL
- Generate the generic service proxy or stub
- The proxy or stub (or a helper class) contains the necessary code to locate a service implementation by searching a UDDI registry.
- Test the client

Dynamic client with known service type



Securing Web-Services

- WS-Security is a message-level security, which means that we can apply various scenarios of WS-Security according to the characteristics of each Web service application.
- For example, to verify who requests the service, we can add an authentication mechanism by inserting various types of security tokens.
- To keep the integrity or confidentiality of the message, digital signatures and encryption are typically applied.

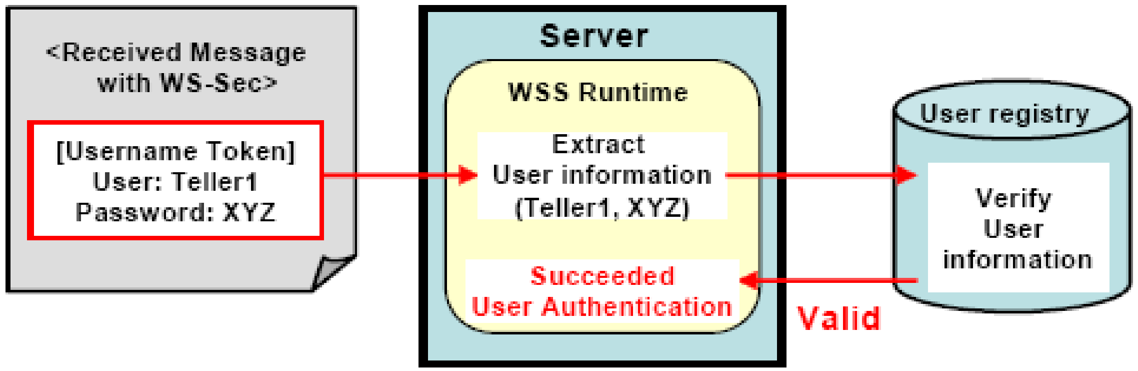
WS-Security:

  - **Kerberos using WS-Security,**
  - **WS-Trust**
  - **WS-Secure Conversation.**

Authentication

To apply an authentication mechanism, it is necessary to insert a security token into the request message of a client.

Web Sphere Application Server Version 6.0 has a pluggable token architecture, which enables a user to implement a custom token.

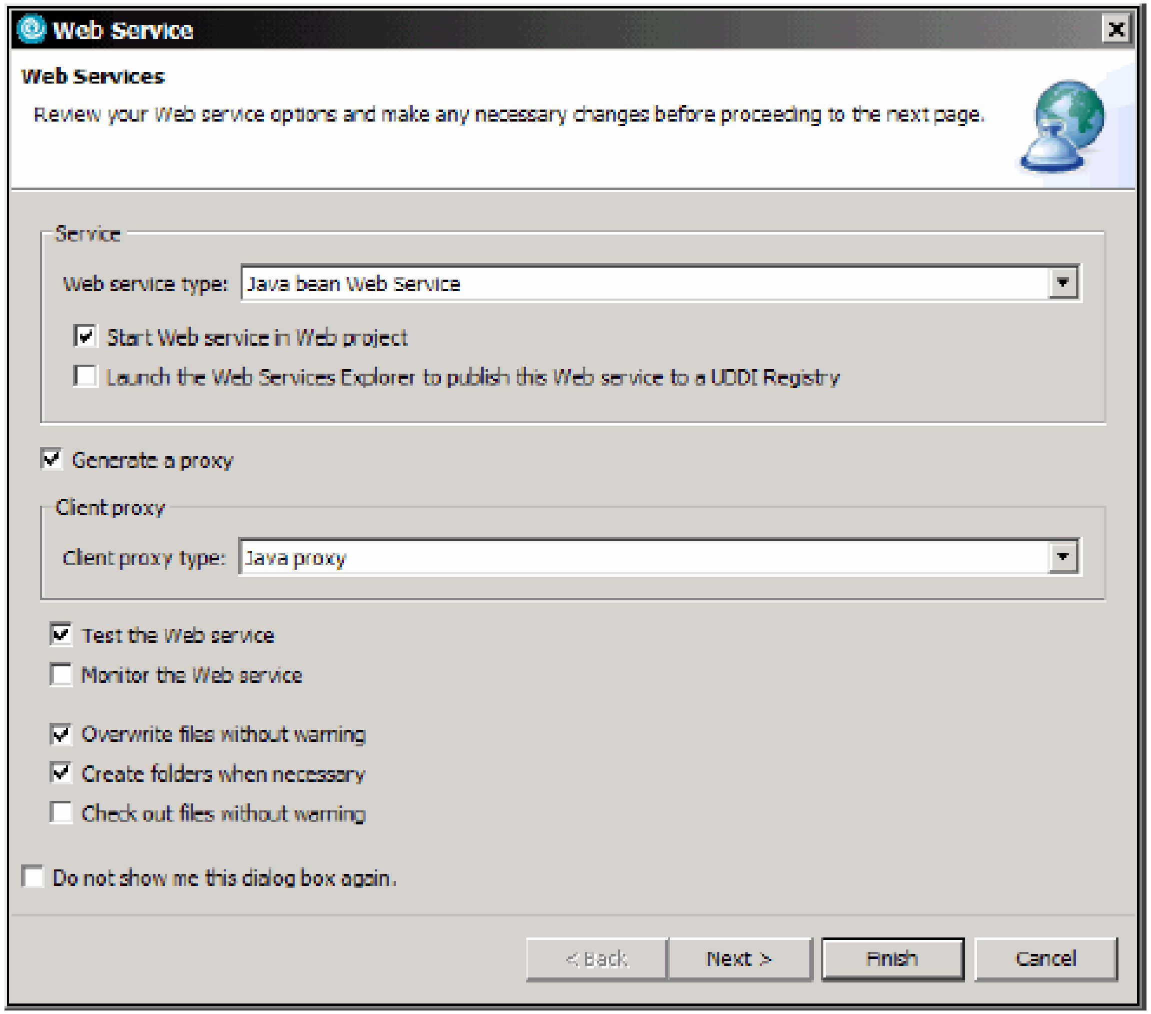


<Received Message with WS-Sec>
[Username Token] User: Teller1 Password: XYZ
Server
WSS Runtime
Extract User information (Teller1, XYZ)
Succeeded User Authentication
Valid
User registry
Verify User information

Web Service wizard

- Switch to the Web perspective Project Explorer.
- Navigate to the Java Bean by expanding Dynamic Web Projects
- WeatherJavaBeanWeb → Java Resources → Java Source → com.hi.hsbc.services
- Select the *WheatherBean*.
- From the context menu, select Web Services → Create Web service.

Web Services page



Web Service

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

Service

Web service type: Java bean Web Service

- [✓] Start Web service in Web project
- [ ] Launch the Web Services Explorer to publish this Web service to a UDDI Registry

- [✓] Generate a proxy

Client proxy

Client proxy type: Java proxy

- [✓] Test the Web service
- [ ] Monitor the Web service

- [✓] Overwrite files without warning
- [✓] Create folders when necessary
- [ ] Check out files without warning

- [ ] Do not show me this dialog box again.

Web Services page

Web service type:

DADX Web Service

Create a Web service from a stored procedure or SQL statement.

Skeleton Java bean Web Service

Create a skeleton from a WSDL file.

EJB Web Service

Create a Web service from an enterprise bean.

Skeleton EJB Web Service

Create a session EJB skeleton from a WSDL file.

Web Services page

Java bean Web Service

- Create a Web service from a Java Bean.

ISD Web Service

- Create a Web service from a Web service deployment descriptor.

URL Web Service

- Create a Web service that returns the content from a URL (for example, a servlet).

Web Services page

Select *Start Web service in Web project*.

Clear *Launch Web Services Explorer to publish this Web service to a UDDI*

Select *Generate a proxy*.

For Client proxy type, select *Java proxy*.

Select *Test the Web service*.

Clear *Monitor the Web service*.

Select *Overwrite files without warning*,

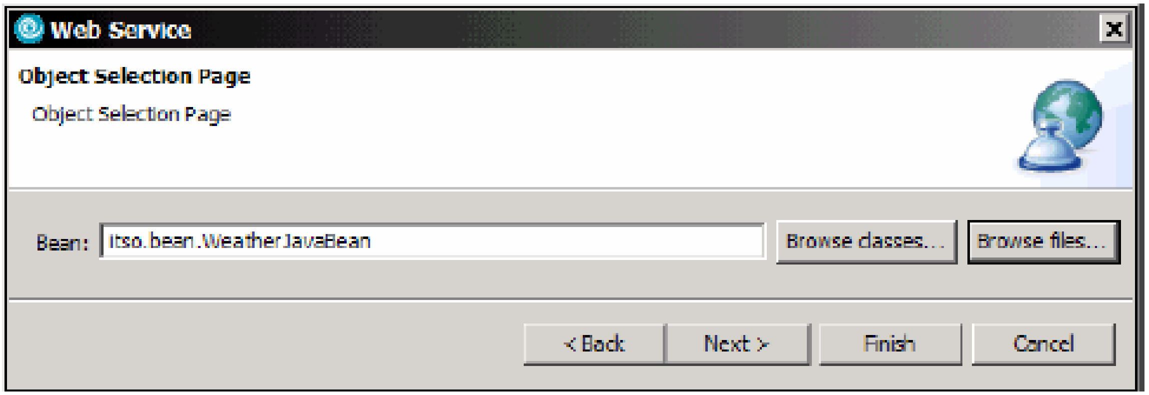
Select *Create folders when necessary*.

Clear *Check out files without warning*.

Click *Next* to proceed to next page.

Object Selection page

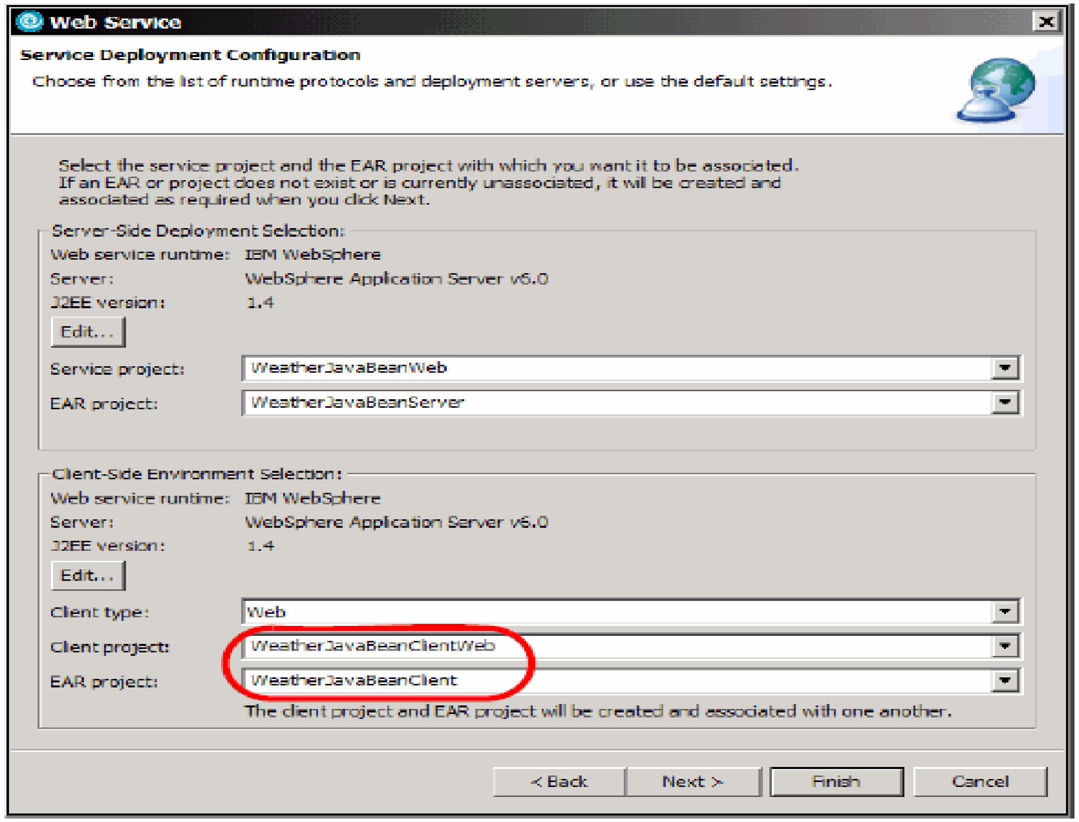
On the Object Selection page, you can specify from which JavaBean the Web service is generated



Bean: itso.bean.WeatherJavaBean

Service Deployment Configuration page

On the Service Deployment Configuration page, specify the deployment settings for the service and the generated test client



Web Service

Service Deployment Configuration

Choose from the list of runtime protocols and deployment servers, or use the default settings.

Select the service project and the EAR project with which you want it to be associated.
If an EAR or project does not exist or is currently unassociated, it will be created and associated as required when you click Next.

Server-Side Deployment Selection:

Web service runtime:	IBM WebSphere
Server:	WebSphere Application Server v6.0
J2EE version:	1.4
Service project:	WeatherJavaBeanWeb
EAR project:	WeatherJavaBeanServer

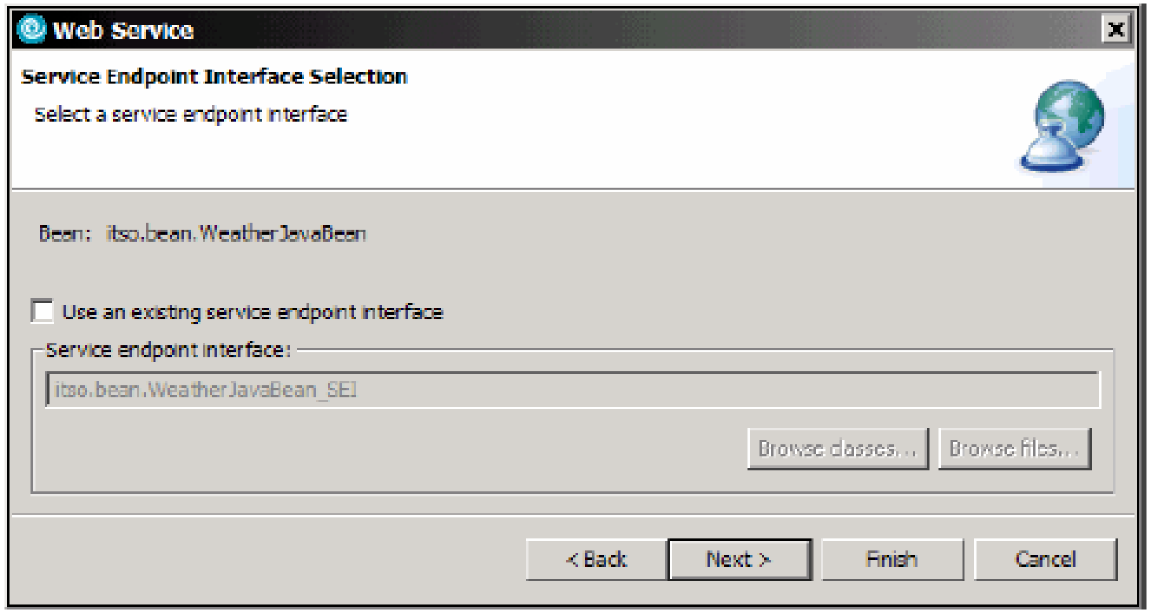
Client-Side Environment Selection:

Web service runtime:	IBM WebSphere
Server:	WebSphere Application Server v6.0
J2EE version:	1.4
Client type:	Web
Client project:	WeatherJavaBeanClientWeb
EAR project:	WeatherJavaBeanClient

The client project and EAR project will be created and associated with one another.

Service Endpoint Interface Selection page

On this page, it is possible to use an existing service endpoint interface (SEI), but not select the option and have the wizard generate the interface



Service Endpoint Interface Selection

Select a service endpoint interface

Bean: itso.bean.WeatherJavaBean

[ ] Use an existing service endpoint interface

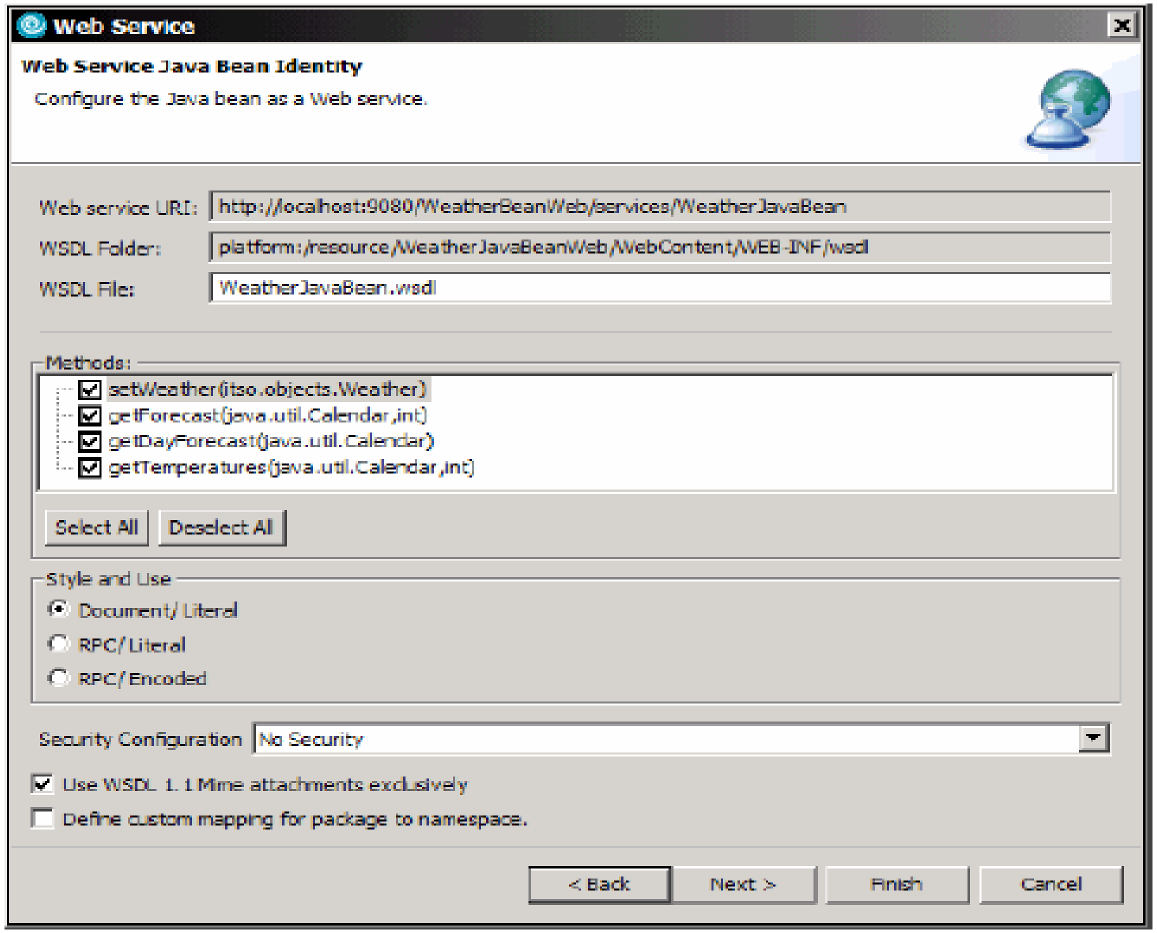
Service endpoint interface: itso.bean.WeatherJavaBean_SEI

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Web Service Java Bean Identity page

- Specify the name of the WSDL file
- Select which methods to expose. In this example, we expose all methods.
- Select Document/Literal for Style And Use, because this is WS-I compliant.
- Select No Security for the Security Configuration.
- Select if custom package to namespace mapping is required. In this example,
- the default mappings are fine.

Web Service Java Bean Identity page



Web Service Java Bean Identity

Configure the Java bean as a Web service.

Web service URI: http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean

WSDL Folder: platform:/resource/WeatherJavaBeanWeb/WebContent/WEB-INF/wsdl

WSDL File: WeatherJavaBean.wsdl

Methods:

- [✓] setWeather(itso.objects.Weather)
- [✓] getForecast(java.util.Calendar,int)
- [✓] getDayForecast(java.util.Calendar)
- [✓] getTemperatures(java.util.Calendar,int)

Style and Use

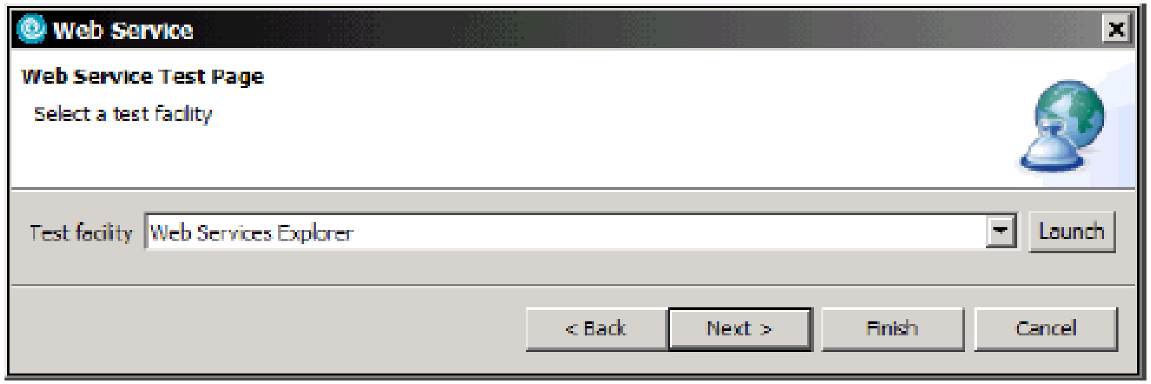
- [● Document/ Literal]
- RPC/ Literal
- RPC/ Encoded

Security Configuration: No Security

- [✓] Use WSDL 1.1 Mime attachments exclusively
- [ ] Define custom mapping for package to namespace.

Web Service Test page

On this page, you can launch the Web Services Explorer to test the Web service before generating the proxy This page is only available if *Test the Web Service* has been selected on the first wizard page.



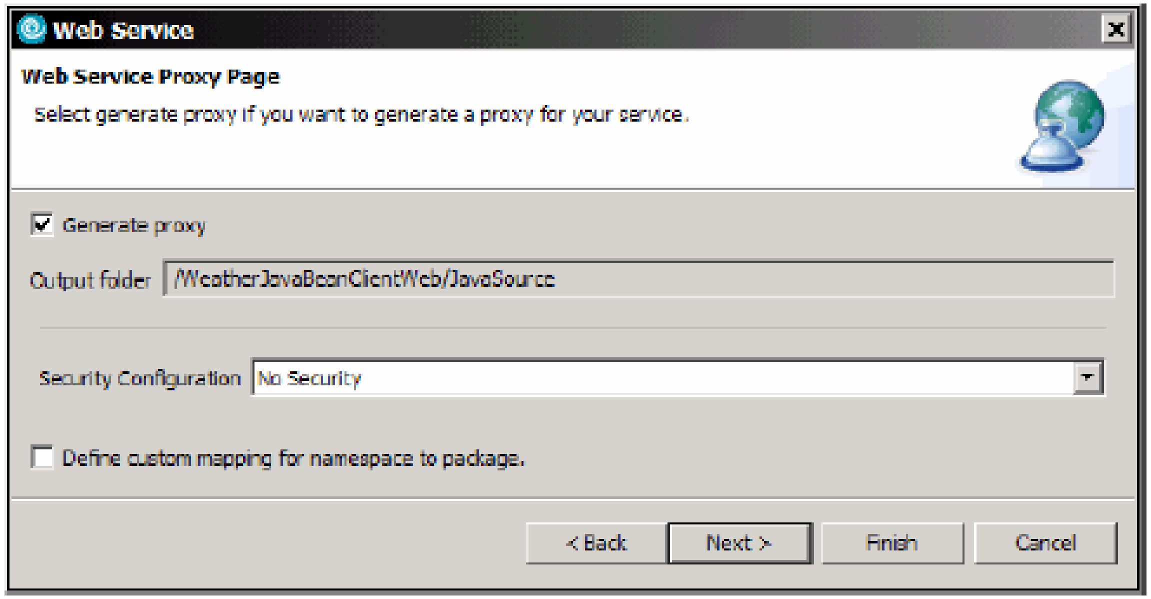
Web Service

Web Service Test Page

Select a test facility

Test facility: Web Services Explorer

Web Service Proxy page



Web Service

Web Service Proxy Page

Select generate proxy if you want to generate a proxy for your service.

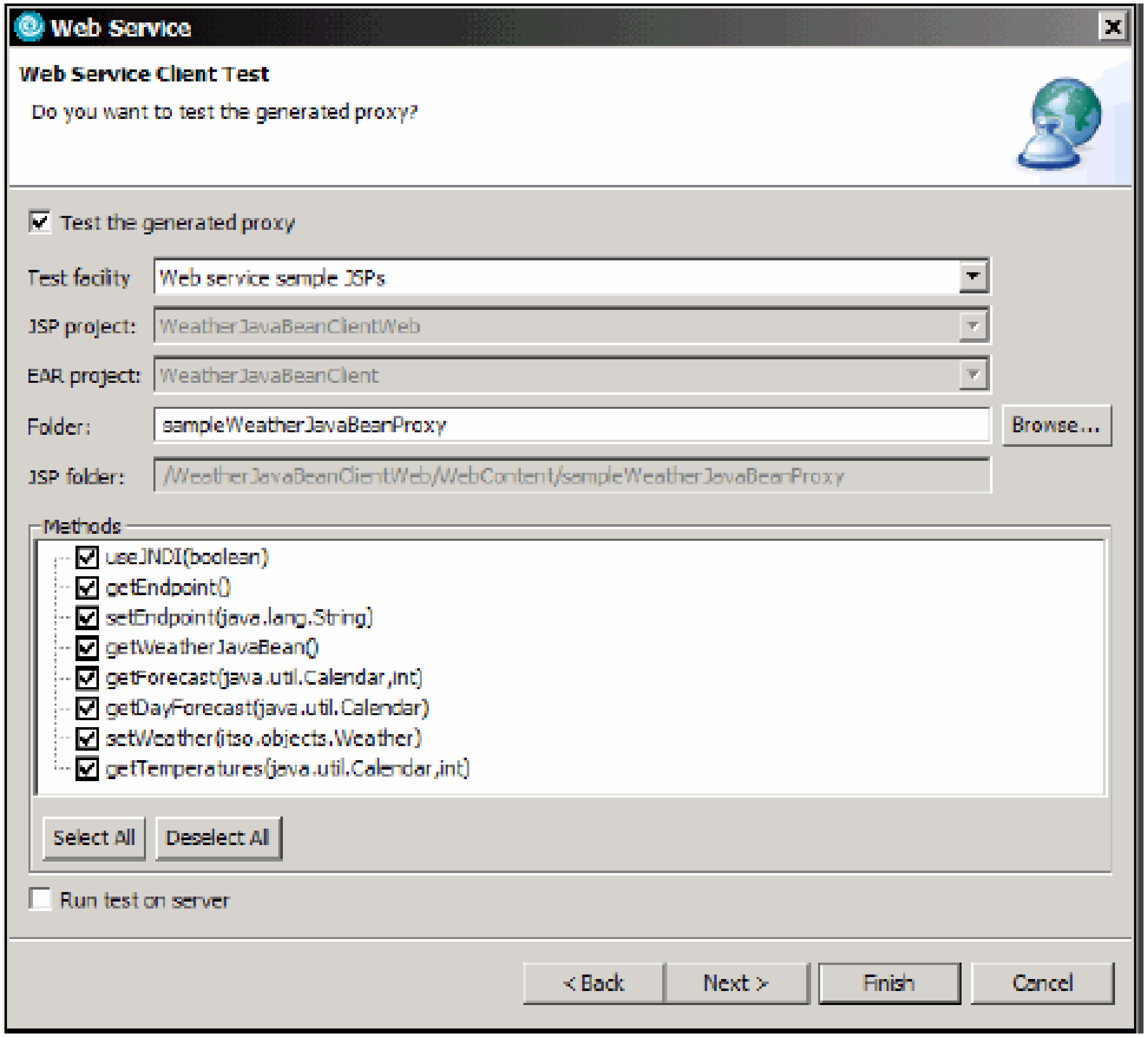
[✓] Generate proxy

Output folder: /WeatherJavaBeanClientWeb/JavaSource

Security Configuration: No Security

[ ] Define custom mapping for namespace to package.

Web Service Client Test page



Web Service

Web Service Client Test

Do you want to test the generated proxy?

[✓] Test the generated proxy

Test facility	Web service sample JSPs
JSP project:	WeatherJavaBeanClientWeb
EAR project:	WeatherJavaBeanClient
Folder:	sampleWeatherJavaBeanProxy
JSP folder:	/WeatherJavaBeanClientWeb/WebContent/sampleWeatherJavaBeanProxy

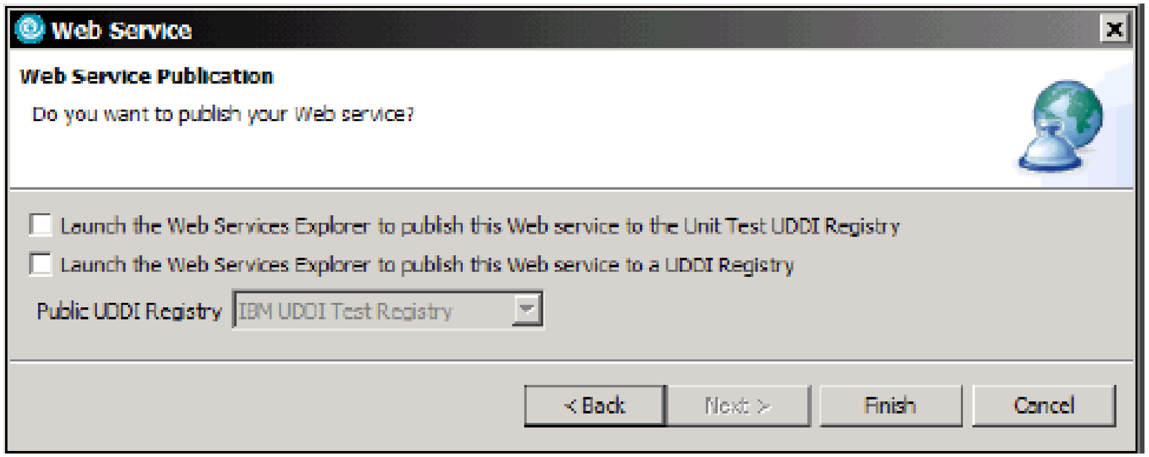
Methods

- [✓] useJNDI(boolean)
- [✓] getEndpoint()
- [✓] setEndpoint(java.lang.String)
- [✓] getWeatherJavaBean()
- [✓] getForecast(java.util.Calendar,int)
- [✓] getDayForecast(java.util.Calendar)
- [✓] setWeather(itso.objects.Weather)
- [✓] getTemperatures(java.util.Calendar,int)

[ ] Run test on server

Web Service Publication page

Leave both options cleared on this page, because the weather service will not be published to an UDDI registry



Web Service

Web Service Publication

Do you want to publish your Web service?

☐ Launch the Web Services Explorer to publish this Web service to the Unit Test UDDI Registry

☐ Launch the Web Services Explorer to publish this Web service to a UDDI Registry

Public UDDI Registry: IBM UDDI Test Registry

Generated files

Files generated in the server project

- Service endpoint interface (SEI): WeatherBean_SEI.java is the interface defining the methods exposed in the Web service.
- WSDL file: /WebContent/WEB-INF/wsdl/WeatherJavaBean.wsdl describes the web service. A copy of this file is also placed in the WebContent/wsdl folder.
- Deployment descriptor: webservices.xml, ibm-webservices-ext.xml and ibm-webservices-bnd.xml. These files describe the Web service according to the Web services for J2EE style
- The JAX-RPC mapping is described in the WeatherJavaBean_mapping.xml file.

Files generated in the client project

Proxy classes

WeatherBean

This is a copy of the service endpoint interface (SEI) containing the methods of the Web service.

WeatherBeanServiceLocator

This class contains the address of the Web service (http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean) and methods to retrieve the address and the SEI:

getWeatherJavaBeanAddress()

getWeatherJavaBean()

getWeatherJavaBean(URL)

WeatherJavaSoapBindingStub

This class implements the SEI for the client. An instance of this class is returned by the getWeatherJavaBean methods of the locator.

WeatherJavaServiceInformation

This class contains descriptive information about the Web service and is not used at execution.

WeatherJavaBeanProxy—This is a helper class that includes the APIs of

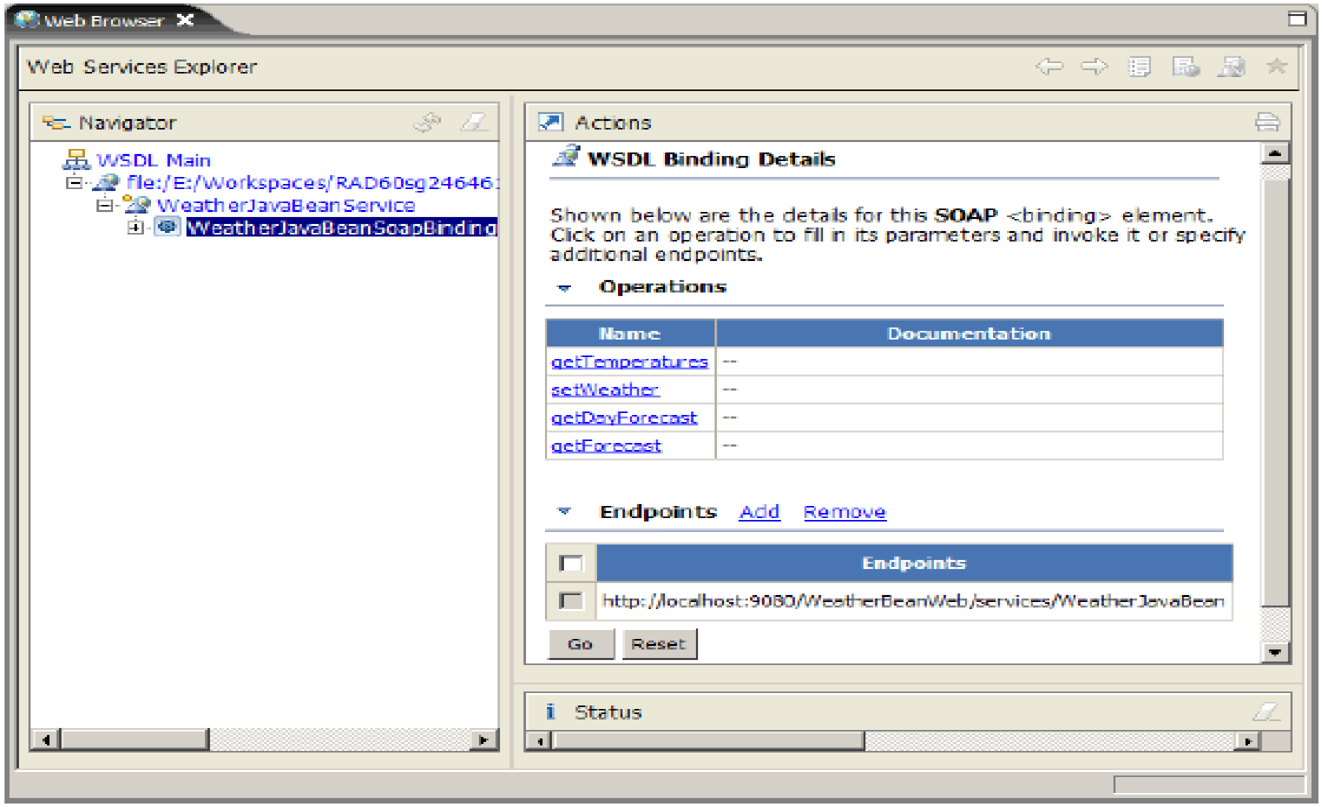
JSR 101 and JSR 109 to get the SEI from the locato

Testing the Web service

- Web Services Explorer
- Test client JSPs
- Universal Test Client

Testing with the Web Services Explorer

To start the Web Services Explorer, select the WeatherJavaBean.wsdl file (in WeatherJavaBeanWeb/WEB-INF/wsdl) and *Web Services → Test with Web Servcies Explorer*



WSDL Binding Details

Shown below are the details for this **SOAP** <binding> element. Click on an operation to fill in its parameters and invoke it or specify additional endpoints.

Operations

Name	Documentation
getTemperatures	--
setWeather	--
getDayForecast	--
getForecast	--

Endpoints Add Remove

☐	Endpoints
☐	http://localhost:9080/WeatherBeanWeb/services/WeatherJavaBean

Go Reset

Testing with the test client JSPs

- Select the TestClient.jsp (in WebContent/sampleWeatherJavaBeanProxy) and
- Run → Run on Server (context).
- When prompted, select the WebSphere Application Server v6.0 and Set
- server as project default. Click Finish.
- The test client opens in a Web Browser Select a method,
- enter the parameter or parameters, click Invoke, and the results are displayed.

Creating Web service clients

Stand-alone Java client