SCDJWS 5 Study Notes

by

Ivan A Krizsan

Version: October 21, 2009

Table of Contents

Purpose

This document contains the notes I made when preparing for the SCDJWS 5 (Sun Certified Developer of Java Web Services) certification.

Licensing

Disclaimers

Though I have done my best to avoid it, this document might contain errors. I cannot be held responsible for any effects caused, directly or indirectly, by the information in this document – you are using it on your own risk.
Submitting any suggestions, or similar, the information submitted becomes my property and you give me the right to use the information in whatever way I find suitable, without compensating you in any way.
All trademarks in this document are properties of their respective owner and do not imply endorsement of any kind.
This document has been written in my spare time and has no connection whatsoever with my employer.

Thanks

Thanks to the following persons that have provided feedback or contributed in some other form to this document:

- Adam Smolnik – Finding numerous mistakes in the first few sections.
- Everyone that has asked questions about this document and its contents.

1. XML Web Service Standards

1.1 XML Documents, W3C Schema and the WS-I Basic Profile 1.1

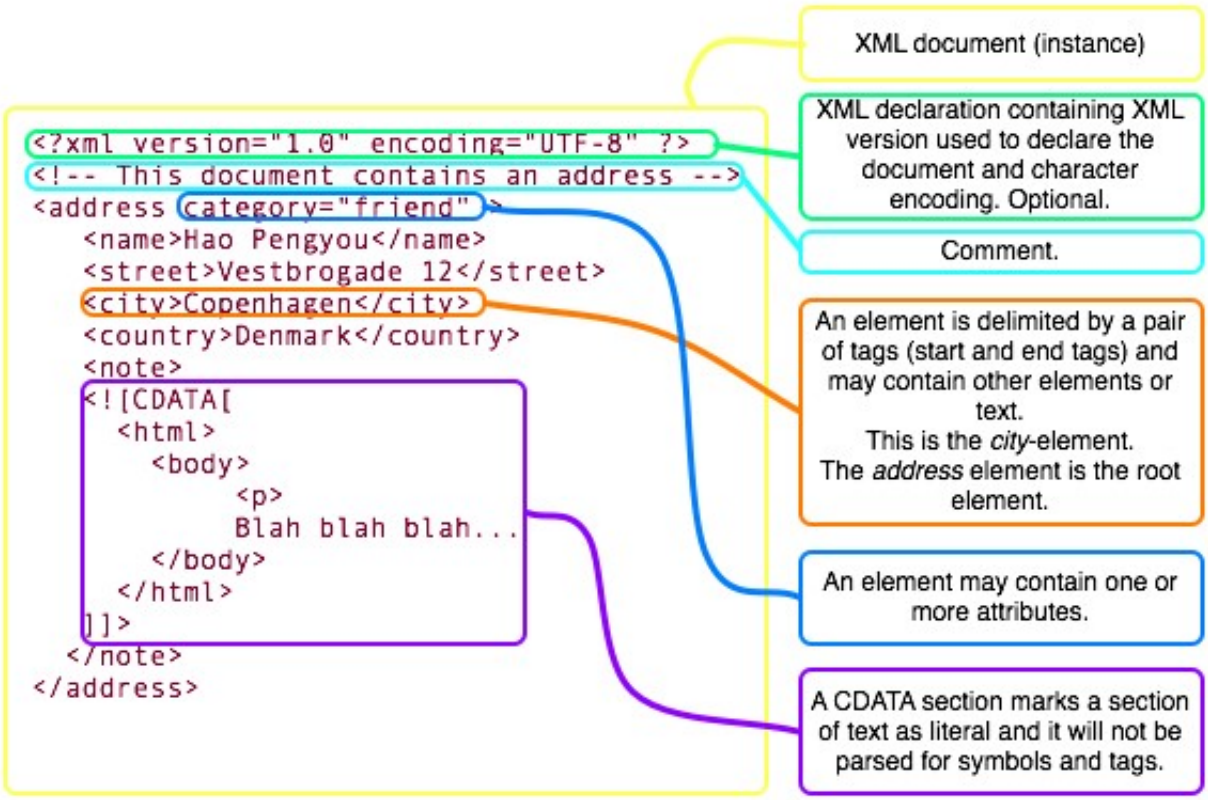
Given XML documents, schemas, and fragments determine whether their syntax and form are correct (according to W3C schema) and whether they conform to the WS-I Basic Profile 1.1.

References:
XML 1.0: http://www.w3.org/TR/2006/REC-xml-20060816/
XML 1.1: http://www.w3.org/TR/2006/REC-xml11-20060816/
http://www.ws-i.org/Profiles/BasicProfile-1.1.html

XML Documents

An XML document represents one possible set of data for a given XML schema.

Structure



- XML documents are composed of Unicode text.
  The WS-I Basic Profile 1.1 stipulates that XML documents for web services must use UFT-8 or UTF-16 encoding.
- An XML document (instance) represents one set of possible data for a particular markup language.
- Each XML document must have one root element that must contain all the other elements and text, except the XML declaration, comments, and certain processing instructions.

Elements

- An element name must begin with a letter or underscore but must not start with the characters “xml” and must not contain any of the following characters: /, <, >, ?, ", @, &, : (and some additional characters).
- Elements may be empty, in which case they look like this:
  <bean-name id=”someName”/> or <street></street>
  Note that an element with attribute(s) may still be empty.
- A CDATA section allows an element to contain literal text that will not be parsed for symbols and tags.
  Example: <![CDATA[text here]]>

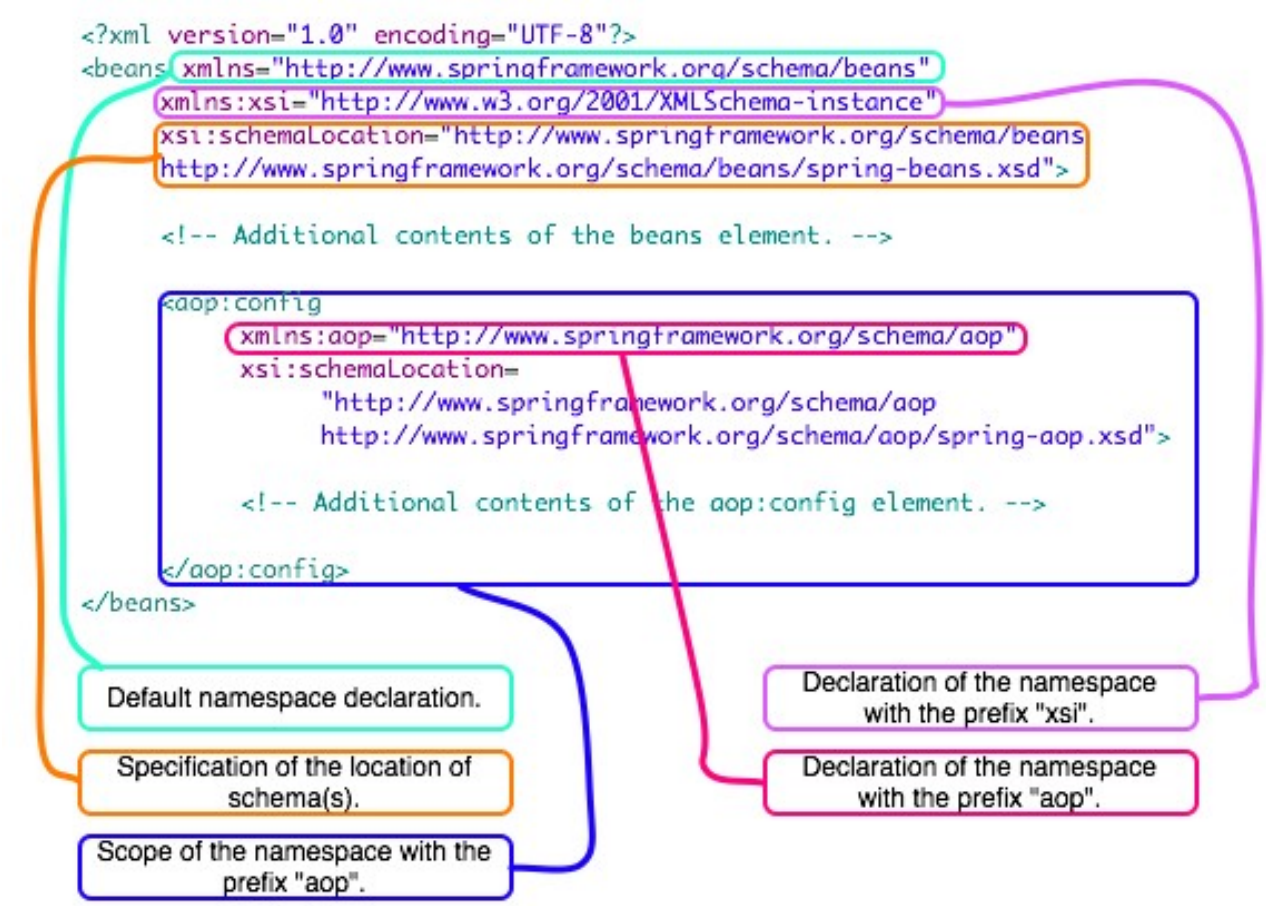
Attributes of Elements

- An element may have zero, or more, attributes.
- An attribute of an element consists of a name-value pair.
- Attribute names have the same restrictions as element names, that is:
  It must begin with a letter or underscore but must not start with the characters “xml” and must not contain any of the following characters: /, <, >, ?, ", @, &, : (and some additional characters).
- The value must be in single or double quotes.
- An attribute may only appear once in an element.
- Attributes must be declared in the start tag (never in the end tag).
- Attributes can not be nested.

Example of an element with three attributes:

```
<phone countrycode=”886” areacode=”03” number=”123456”/>
```

Namespaces



Continued on next page...

- A XML namespace is declared using the *xmlns* attribute in the following form:
  <myTag xmlns=”someURI” ...>
  The URI must conform to the URI specification. The most common form is the URL.
- Each namespace must have an unique identifier specified by its URI.
- The URI does not have to point to an actual resource.
- The default namespace is defined as above:
  <myTag xmlns=”someURI” ...>
- A namespace can be assigned a prefix that is used to fully qualify elements of the namespace.
  Example: <myTag xmlns:my=”someURI” ... >
- The scope of a namespace declaration, whether it is a default namespace declaration or a namespace declaration declaring a prefix, ranges from the beginning of the start-tag in which it appears to the end of the corresponding end-tag.
  The scope excludes any elements overriding any namespace(s) by declaring namespace(s) with the same prefix (or the default namespace).
- Schema locations are specified by one or more pairs of [namespace, physical location].
  The physical URL location.
  Schema locations are only a hint to the XML schema parser – it does not have to use it.
- If an XML schema does not specify *elementFormDefault="qualified"*, then the child elements are automatically assumed to be in the namespace of the parent.
- For example: If the aop XML schema used in the figure above does not require qualified elements, then the child elements does not need to be qualified, even though they belong to the aop namespace.
- An XML namespace declaration xmlns=”” cancels the default namespace.

XML Schemas

An XML schema describes a markup language; defining which elements and attributes are used in the language, their structure and what their data-types are. XML schemas are XML documents.

The schema Element

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:ivan="http://www.ivan.com/schemas"
      targetNamespace="http://www.ivan.com/schemas"
       attributeFormDefault="qualified" elementFormDefault="qualified">

      <!-- Additional schema contents. -->
</schema>
```

The root element of an XML schema is always the <schema> element. The <schema> element in the above example makes the following declarations:

- It assigns the XML schema namespace “http://www.w3.org/2001/XMLSchema” as the default namespace.
  This namespace is the standard namespace defined by the XML schema specification. All the XML schema elements must belong to this namespace – this in order, for instance, to gain access to the predefined simple types such as “string”, “int” etc. etc.
- Using an XML namespace declaration, it assigns the prefix “ivan” to the namespace “http://www.ivan.com/schemas”.
  New types created in the schema can be referred to using this prefix.
  For example, if a type age has been declared in the schema, it can be used like this:
  <element name="age" type="ivan:age" />
- It declares the target namespace as being “http://www.ivan.com/schemas”.
  All new types explicitly created within the schema will belong to this namespace.
- The attribute *attributeFormDefault* which specifies whether attributes need to be qualified by the prefix of the namespace to which they belong or not.
  The default value is “unqualified”.
- The attribute *elementFormDefault* which specifies whether elements need to be qualified by the prefix of the namespace to which they belong or not.
  The default value is “unqualified”.

If the attribute *elementFormDefault* is set to “qualified” in the schema declaration, like in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    elementFormDefault="qualified">

    ...
</schema>
```

...then the local elements in the target namespace, element(s) in bold, must be qualified by the namespace prefix, like in this XML document:

```
<ivan:person xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas personSchema.xsd"
    hasDog=”true”>

    <ivan:firstName>Steven</ivan:firstName>
    <ivan:lastName>Segal</ivan:lastName>
    <ivan:age>39</ivan:age>
</ivan:person>
```

If, on the other hand, the attribute *elementFormDefault* is set to “unqualified” in the schema declaration, or if the *elementFormDefault* attribute is omitted, then the local elements must not be qualified by the namespace prefix, like in this XML document:

```
<ivan:person xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas personSchema.xsd"
    hasDog=”true”>

    <firstName>Steven</firstName>
    <lastName>Segal</lastName>
    <age>39</age>
</ivan:person>
```

Note! Global element(s) of the target namespace must always be fully qualified.

If the attribute *attributeFormDefault* is set to “qualified” in the schema declaration, like in this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="qualified">

    ...
</schema>
```

...then the local attributes of the target namespace, attribute(s) in bold, must be qualified by the namespace prefix, like in this XML document:

```
<ivan:person xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas personSchema.xsd"
    ivan:hasDog="true">

    <firstName>Steven</firstName>
    <lastName>Segal</lastName>
    <age>39</age>
</ivan:person>
```

Note! Global attribute(s) of the target namespace must always be fully qualified.

Elements

Element declarations are used to declare elements of an XML schema.

```
...
<element name="elementName" abstract="false" final="#all" default="defaultValue"
nillable="true" type="string"/>
...
```

Element Declaration Attributes

An element declaration can have the following attributes (list not exhaustive):

Attribute Name	Description
abstract	Indicates whether the element is abstract or not. Values: true|false Default: false
default	Default value of the element.
final	Restrictions on derivation of new types from this type. Empty string means no restrictions (default value). #all means that no derivations may be done.
form	Whether the element needs to be qualified with a namespace prefix or not. Values: qualified|unqualified Default: unqualified
maxOccurs	Maximum number of occurrences of the element. Values: [non negative integer]|unbounded Default: 1
minOccurs	Minimum number of occurrences of the element. Values: [non negative integer] Default: 1
name	Name of the element.
nillable	Can the element contain nil? Values: true|false Default: false
type	Type of the element. Cannot be used if the <element> element contains a <simpleType> or a <complexType> element.

Element Type

An element declaration can, if no *type* attribute specified, contain a simple or complex type declaration.
The value of *type* attributes can be either one of the 44 built-in types, which belong to the XML schema namespace "http://www.w3.org/2001/XMLSchema", or a complex type. The built-in types can only contain data and cannot contain other elements.

Global Elements

In order for an XML instance document to be able to directly refer to an element declared in an XML schema, the element must be declared as a global element. Global elements are declared as immediate children of the <schema> element, as compared to children of a complex type. Any number of global elements may be declared in an XML schema.
In this example, the person type can be used in the following XML document.
The following XML schema is stored in the file “personSchema.xsd”.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas">

    <!--
       This is a declaration of a global element which makes it possible
       to use the element in the root of an XML document (instance).
     -->
    <element name="person" type="ivan:Person"/>

    <complexType name="Person">
        <sequence>
            <element name="firstName" type="string" nillable="false"/>
            <element name="lastName" type="string" nillable="false"/>
            <element name="age" type="int"/>
            <element name="favColour" type="string" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
    </complexType>

</schema>
```

This example shows a possible instance of the above XML schema:

```
<ivan:person xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas personSchema.xsd">

    <firstName>Steven</firstName>
    <lastName>Segal</lastName>
    <age>39</age>
</ivan:person>
```

The following is an example of an XML schema that only contains a single element declaration and an instance of that schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:ivan="http://www.ivan.com/element"
      targetNamespace="http://www.ivan.com/element">

      <element name="myElement" type="date"/>
</schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:myElement
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ivan="http://www.ivan.com/element"
      xsi:schemaLocation="http://www.ivan.com/element elementSchema.xsd">
      2008-09-14
</ivan:myElement>
```

Complex Types

An XML schema may declare complex type which describe how elements that contain other elements are organized. The content pattern for the <complextType> element is:

```
annotation?
(simpleContent |
    complexContent |
    ((group | all | choice | sequence)?
        (attribute | attributeGroup)* anyAttribute?)))
```

An example of a declaration of a <complexType> in a XML schema:

```
    ...
    <complexType name="Person" abstract="true">
        <sequence>
            <element name="firstName" type="string" nillable="false" />
            <element name="lastName" type="string" nillable="false" />
            <element name="age" type="int"/>
        </sequence>
        <attribute name="type" type="string" use="optional" default="ordinary"/>
    </complexType>
    ...
```

The <complexType> element can have the following attributes (list not exhaustive):

Attribute Name	Description
abstract	Indicates whether the complex type is abstract or not. Values: true|false An abstract type may only be used to specify new types and cannot be instantiated. Default: false
name	The type name used to refer to the complex type.
final	Restricts the creation of sub-types based on this type. Values: #all (restriction and extension), extension, restriction.

The final attribute in the <complexType> element can have one of the following values:

- #all
  Prevents any kind of sub-typing of the complex type.
- extension
  Prevents extension, but not restriction, of the complex type.
- restriction
  Prevents restriction, but not extension, of the complex type.

The <sequence> Element

The above example of a complex type declares a type that:

- Has the name “Person”.
  Type names are case sensitive.
- Is abstract.
- Can contain the elements “firstName”, “lastName” and “age” in that sequence.
  Elements can be optional or appear multiple times within a sequence.

Has one attribute that specifies which type of person is described.
This attribute is a string, is optional and defaults to “ordinary”.
Example of an instance of the above “Person” type:

```
…
<person type=”close friend”>
    <firstName>Johnny</firstName>
    <lastName>Mnemonic</lastName>
    <age>32</age>
</person>
…
```

The <all> Element

Reference: http://www.w3.org/TR/xmlschema-1/#Model_Groups

Complex types can be declared containing the <all> element in the place of the <sequence> element. The <all> element cause the elements of the complex type to:

- Child elements of the complex type can appear in any order.
- Child elements always have a *maxOccurs* of 1 and a *minOccurs* of 0.
- Only single elements may occur in an <all> group.
  No groupings like <sequence> or <all> can occur in an <all> group, however, custom types declared using either <simpleType> or <complexType> can occur in an <all> group.

If we have the following <complexType> declaration that uses the <all> element:

```
...
<complexType name="House">
    <all>
        <element name="street" type="string"/>
        <element name="city" type="string"/>
        <element name="inhabitants" type="h:Persons"/>
    </all>
</complexType>
...
```

Then an XML document containing a house may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<h:house
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:h="http://www.ivan.com/schemas"
      xsi:schemaLocation="http://www.ivan.com/schemas houseSchema.xsd">

      <city>Gothenbug</city>
      <inhabitants>
            <!-- List of inhabitants here, see other example for details. -->
      </inhabitants>
      <street>Old Road 1</street>
</h:house>
```

Notice that the elements of the house does not appear in the order they were declared in the XML schema.

The <choice> Element

The <choice> element allows one of the elements contained in the <choice> element to be present in the containing element.
Given the following XML schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:chex="http://www.ivan.com/choiceExample"
    targetNamespace="http://www.ivan.com/choiceExample">

    <element name="computer-owner" type="chex:ComputerOwner"/>

    <complexType name="ComputerOwner">
        <sequence>
            <element name="name" type="string"/>
            <choice minOccurs="1" maxOccurs="2">
                <element name="desktop" type="string"/>
                <element name="notebook" type="string"/>
                <element name="handheld" type="string"/>
            </choice>
        </sequence>
    </complexType>
</schema>
```

The following is a valid XML document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<brap:computer-owner
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:brap="http://www.ivan.com/choiceExample"
    xsi:schemaLocation="
        http://www.ivan.com/choiceExample choiceExampleSchema.xsd">

    <name>Ivan</name>
    <notebook>MacBook</notebook>
    <desktop>Big Bertha</desktop>
</brap:computer-owner>
```

The <simpleContent> Element

The <simpleContent> element enables adding of attributes to global simple types. This element can also be used to extend or restrict attributes on other complex types with simple content.
Example of a XML schema declaring a <complexType> element containing a <simpleContent> element:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.ivan.com/PriceWithCurrencySchema"
    xmlns:tns="http://www.ivan.com/PriceWithCurrencySchema"
    elementFormDefault="qualified">

    <element name="PriceAndCurrency" type="tns:PriceWithCurrency"/>

    <complexType name="PriceWithCurrency">
        <simpleContent>
            <extension base="decimal">
                <attribute name="currency" type="string"/>
            </extension>
        </simpleContent>
    </complexType>
</schema>
```

Example of a corresponding XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PriceAndCurrency
    xmlns:tns="http://www.ivan.com/PriceWithCurrencySchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.ivan.com/PriceWithCurrencySchema
        PriceWithCurrencySchema.xsd"
    currency="USD">
        168.4
</tns:PriceAndCurrency>
```

Attributes of Complex Types

The <attribute> element that may appear in the declaration of a complex type can have the following attributes (list not exhaustive):

Attribute Name	Description
default	Default value.
form	Whether the attribute needs to be qualified with a namespace prefix or not. Values: qualified|unqualified Default: unqualified
name	Name of the attribute.
type	Type of the attribute.
use	How the attribute can be used. That is, if it's use is optional, required or prohibited. Values: optional|prohibited|required Default: optional

Inheritance of Complex Types

XML schema supports inheritance of complex types in a manner similar to inheritance found in object-oriented programming. There are two types of complex type inheritance:

- Extension
  An extended type inherits the elements and attributes of the base type and adds new ones.
- Restriction
  The extended type redefines the elements and/or attributes of the base type it wishes to retain and omits the ones it does not want to retain.

The following examples are all declared within one and the same schema document.

Extension

The first example shows how the Policeman type extends the Person type by adding some new elements (precinct, precinctCode and captain) and a new attribute (uniformSize).

```
...
<!-- The Person type is made abstract, in order to prevent direct use. -->
<complexType name="Person" abstract="true">
    <sequence>
        <element name="firstName" nillable="false" type="string"/>
        <element name="lastName" type="string" nillable="false" />
        <element name="age" type="ivan:age" />
    </sequence>
    </complexType>

    <complexType name="Policeman">
        <complexContent>
        <!-- A policeman extends a person. -->
        <extension base="ivan:Person">
            <!-- Here we add the elements specific to a policeman. -->
            <sequence>
                <element name="precinct" type="string"/>
                <element name="precinctCode" type="string"/>
                <!--
                    We need to set minOccurs to 0 in order to be able
                    to omit the element when restricting the Policeman
                    type.
                -->
                <element name="captain" type="string" minOccurs="0"/>
            </sequence>
            <attribute name="uniformSize" type="integer" use="required"/>
        </extension>
    </complexContent>
</complexType>
...
```

An example of an XML document containing a policeman can look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:policeman
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ivan="http://www.ivan.com/schemas"
    xsi:schemaLocation="http://www.ivan.com/schemas policemanSchema.xsd"
    uniformSize="12">

    <firstName>Jack</firstName>
    <lastName>Killian</lastName>
    <age>38</age>
    <precinct>SF Blue 3rd</precinct>
    <precinctCode>AA1234-Z9</precinctCode>
    <captain>Carl Zymak</captain>
</ivan:policeman>
```

Restriction

In the next example, the PoliceCaptain type restricts the Policeman type by defining only the elements that are to be used with a police captain, leaving out the <captain> element from the Policeman type. Note that in order to be able to leave it out, it must have set the attribute *minOccurs* to zero, as shown above.

```
...
<!--
    The PoliceCaptain type is made final with regard to extension,
    that is it can not be extended, but still be restricted.
-->
<complexType name="PoliceCaptain" final="extension">
    <complexContent>
        <!-- A police captain is a police man, but has no captain. -->
        <restriction base="ivan:Policeman">
            <!--
                Here we declare all the elements that a police captain is
                to contain, leaving out or redefining as desired.
            -->
            <sequence>
                <element name="firstName" type="string"/>
                <element name="lastName" type="string"/>
                <!--
                    Note that even if the captain would prefer, we cannot make
                    age optional (minOccur="0") since a child type cannot be less
                    restrictive than the parent type, which in this case requires
                    the age element to be present.
                -->
                <element name="age" type="ivan:age"/>
                <element name="precinct" type="string"/>
                <element name="precinctCode" type="string"/>
            </sequence>
            <!--
                Even though the attribute uniformSize is not listed here,
                it will be required in instances of PoliceCaptain.
            -->
        </restriction>
    </complexContent>
</complexType>
...
```

Corresponding to the discussion on the <captain> element above, the attribute *uniformSize* cannot be made prohibited like this:

```
…
<!-- This will result in a validation error. -->
<attribute name="uniformSize" type="integer" use="prohibited"/>
...
```

Since the base type requires the *uniformSize* attribute, the PoliceCaptain type cannot prohibit the use of this attribute.

An example of an XML document containing a police captain can look like this:

```
<?xml  version="1.0" encoding="UTF-8"?>
<ivan:policeCaptain
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ivan="http://www.ivan.com/schemas"
    xsi:schemaLocation="http://www.ivan.com/schemas policemanSchema.xsd"
    uniformSize=”22”>

    <firstName>Carl</firstName>
    <lastName>Zymak</lastName>
    <age>46</age>
    <precinct>SF Blue 3rd</precinct>
    <precinctCode>AA1234-Z9</precinctCode>
</ivan:policeCaptain>
```

Polymorphism and Abstract Base Types

As object-oriented programming can assign objects of a subtype to a variable of an abstract super-type, so can subtypes be used with elements of the base type.
First the XML schema definition; notice that the Person type is abstract and that Policeman, Doctor and Carpenter are subtypes of Person. Also notice that the Persons type define an element that contains zero or more Persons.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:h="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas">


    <element name="house" type="h:House"/>
    <element name="policeman" type="h:Policeman"/>
    <element name="doctor" type="h:Doctor"/>
    <element name="carpenter" type="h:Carpenter"/>

    <!-- The Person type is made abstract, in order to prevent direct use. -->
    <complexType name="Person" abstract="true">
        <sequence>
            <element name="firstName" nillable="false" type="string"/>
            <element name="lastName" type="string" nillable="false" />
            <element name="age" type="int" />
        </sequence>
    </complexType>

    <complexType name="Policeman">
        <complexContent>
            <extension base="h:Person">
                <sequence>
                    <element name="precinct" type="string"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <complexType name="Doctor">
        <complexContent>
            <extension base="h:Person">
                <sequence>
                    <element name="speciality" type="string"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <complexType name="Carpenter">
        <complexContent>
            <extension base="h:Person">
                <sequence>
                    <element name="favouriteHammer" type="string"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

    <complexType name="Persons">
        <sequence>
            <element name="person" type="h:Person" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
    </complexType>

    <complexType name="House">
        <all>
            <element name="street" type="string"/>
            <element name="city" type="string"/>
            <element name="inhabitants" type="h:Persons"/>
        </all>
    </complexType>
</schema>
```

Next, an instance of the above XML schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<h:house
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:h="http://www.ivan.com/schemas"
      xsi:schemaLocation="
            http://www.ivan.com/schemas houseSchema.xsd">

      <city>Gothenbug</city>
      <inhabitants>
            <person xsi:type="h:Carpenter">
                  <firstName>Ted</firstName>
                  <lastName>Nugent</lastName>
                  <age>56</age>
                  <favouriteHammer>PowerFlower Axe</favouriteHammer>
            </person>
            <person xsi:type="h:Doctor">
                  <firstName>Frank</firstName>
                  <lastName>Stein</lastName>
                  <age>238</age>
                  <speciality>Decapitation</speciality>
            </person>
            <person xsi:type="h:Policeman">
                  <firstName>Jack</firstName>
                  <lastName>Killian</lastName>
                  <age>38</age>
                  <precinct>SF Blues</precinct>
            </person>
      </inhabitants>
      <street>Old Road 1</street>
</h:house>
```

Notice how the type of each person living in the house is specified using the *xsi:type* attribute. Leaving the type specification out will cause validation error, since the Person type is abstract.

Simple Types

Apart from the simple types like string, “int”, etc. we have seen, new simple types that further restrict the built-in simple types can be created. This is done in a manner similar to complex type restriction inheritance, as described above.
Simple types differ from complex types in that they cannot contain child elements or attributes. The content pattern for the <simpleType> element is:

```
annotation? (restriction | list | union)
```

The following example, taken from an XML schema declaration, declares the “age” type:

```
...
    <!--
        Define a new simple type used for a person's age.
        This type cannot be further extended (final="#all"),
        has to have a value in the range from 0 to 150.
        White-space in values of the age type will be collapsed,
        i.e. TAB, C/R and LF will be replaced with space, two or more continuous
        occurrences of spaces will be replaced with one single space,
        finally heading and trailing white-space will be removed.
    -->
    <simpleType name="age" final="#all">
        <restriction base="integer">
            <minInclusive value="0"/>
            <maxInclusive value="150"/>
            <whiteSpace value="collapse"/>
        </restriction>
    </simpleType>
...
```

If the age-type is declared in a namespace with the prefix “ivan”, then it can later be used in the following manner:

```
<element name="age" type="ivan:age" />
```

The elements contained in the <restriction> element are called facets. A facet represents a characteristic of the built-in type that is restricted. Please refer to the XML Schema specification for a complete list of which facets are available for the different built-in types.

Facets

This section will show examples of how to use some common facets when defining simple types.

Numerical Facets

When creating the simple type “age”, the facets *minInclusive* and *maxInclusive* were used to restrict the range of the value. The following related facets are available:

Facet	Description
minInclusive	Minimum (inclusive) allowed value.
maxInclusive	Maximum (inclusive) allowed value.
minExclusive	Minimum (exclusive) allowed value.
maxExclusive	Maximum (exclusive) value.

Pattern and Length

The *pattern* facet allows for supplying a regular expression that the value of the simple type must match. Notice that the *pattern* facet is not only available for the string built-in type, but also for numeric types like float etc.
Let's assume we want to restrict the code of a police precinct to a certain pattern and its length to eight characters, the latter by using the *length* facet.

```
…
<simpleType name="precinctCode" final="#all">
    <restriction base="string">
        <length value="8"/>
        <pattern value="[A-D]{2}[0-9]{4}-[A-Z][0-9]"/>
    </restriction>
</simpleType>
...
```

Enumeration

The *enumeration* facet allows for providing a list of allowed values:

```
...
<simpleType name="colour">
    <restriction base="string">
        <enumeration value="Blue"/>
        <enumeration value="Green"/>
        <enumeration value="Pink"/>
        <!-- Etc. -->
   </restriction>
</simpleType>
...
```

Lists and Unions

Simple types may also be lists and unions. A list is a space-separated list of simple values. Values in a list may not contain spaces, line-feeds or tabs. A union specifies several simple types, of which the value of the element having the simple type using the union can be of one of the types declared in the union.

The following schema declares a list type, *myCodeList*, and an union type, *codeNumberUnion*.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:lu="http://www.ivan.com/lisuniSchema"
      targetNamespace="http://www.ivan.com/lisuniSchema">

      <element name="luTest" type="lu:myLUTest"/>

      <simpleType name="myCode">
            <restriction base="string">
                  <enumeration value="AAA"/>
                  <enumeration value="BBB"/>
                  <enumeration value="CCC"/>
            </restriction>
      </simpleType>

      <simpleType name="oneToHundred">
            <restriction base="int">
                  <minInclusive value="1"/>
                  <maxInclusive value="100"/>
            </restriction>
      </simpleType>

      <simpleType name="myCodeList">
            <list itemType="lu:myCode"/>
      </simpleType>

      <simpleType name="codeNumberUnion">
            <union memberTypes="lu:myCode lu:oneToHundred"/>
      </simpleType>

      <complexType name="myLUTest" final="#all">
            <sequence>
                  <element name="code" type="lu:myCode"/>
                  <element name="number" type="lu:oneToHundred"/>
                  <element name="codeList" type="lu:myCodeList"/>
                  <element name="cnUnion" type="lu:codeNumberUnion"
                       maxOccurs="unbounded"/>
            </sequence>
      </complexType>
</schema>
```

Continued on next page...

The following XML document is an instance of the above XML schema and shows some valid and some invalid values for the list and union types.

```
<?xml version="1.0" encoding="UTF-8"?>
<lu:luTest
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lu="http://www.ivan.com/lisuniSchema"
      xsi:schemaLocation="http://www.ivan.com/lisuniSchema listAndUnionSchema.xsd">

      <!-- Valid, since a code is AAA, BBB or CCC. -->
      <code>AAA</code>

      <!-- Valid, since a number must be in the range 1 to 100 (incl). -->
      <number>99</number>

      <!-- Valid, since all the codes in the list are valid. -->
      <codeList>BBB AAA CCC</codeList>

      <!-- Invalid, since the code DDD in the list is invalid. -->
      <!-- <codeList>AAA DDD</codeList> -->

      <!-- Valid, since the union may contain a code or a number. -->
      <cnUnion>100</cnUnion>

      <!-- Valid, since the union may contain a code or a number. -->
      <cnUnion>BBB</cnUnion>

      <!-- Invalid, since the code in the union is invalid. -->
      <!-- <cnUnion>DDD</cnUnion> -->

      <!-- Invalid, since the number in the union is out of range. -->
      <!-- <cnUnion>101</cnUnion> -->
</lu:luTest>
```

Global Attributes

An XML schema can also declare global attributes, which later can be used in the schema declaration(s). This XML schema declares a global attribute “colour”. Notice how it is used to declare an attribute in the complex type Person.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas">

    <element name="person" type="ivan:Person"/>

    <!-- This is a global attribute. -->
    <attribute name="colour" type="string" default="blue"/>

    <complexType name="Person">
        <sequence>
            <element name="firstName" type="string" nillable="false"/>
            <element name="lastName" type="string" nillable="false"/>
            <element name="age" type="int"/>
        </sequence>
        <!-- Here we use the global attribute to add a “colour” attribute to Person. -->
        <attribute ref="ivan:colour"/>
    </complexType>
</schema>
```

...and this XML document shows how the attribute is used:

```
<ivan:person xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas personSchema.xsd"
    ivan:colour="red">

    <firstName>Steven</firstName>
    <lastName>Segal</lastName>
    <age>39</age>
</ivan:person>
```

Note that attributes declared using global attributes always must be fully qualified(?).

The anyAttribute Element

When, for instance, declaring XML schemas for SOAP header blocks, the header block element should allow for the attribute *role* and *mustUnderstand*, declared as global attributes by the SOAP Envelope schema. The attributes can be included in the header block XML schema without directly referencing the global attribute by using the <anyAttribute> element.

```
<?xml version="1.0" encoding="UTF-8"?>
 <schema
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:pb="http://www.krizsan.com/procBy"
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     targetNamespace="http://www.krizsan.com/procBy">

     <import namespace="http://schemas.xmlsoap.org/soap/envelope/"
            schemaLocation="soapEnvelope.xsd"/>

     <element name="processed-by" type="pb:ProcessedBy"/>

     <complexType name="ProcessedByNode">
            <sequence>
                   <element name="time" type="time"/>
                   <element name="id" type="string"/>
            </sequence>
     </complexType>

     <complexType name="ProcessedBy">
            <sequence>
                   <element name="node" type="pb:ProcessedByNode"
                          maxOccurs="unbounded"/>
            </sequence>
            <anyAttribute namespace="##other" processContents="lax"/>
     </complexType>
 </schema>
```

Continued on next page...

The <anyAttribute> element can have the following attributes:

Attribute Name	Description
id	Unique id of the element. Optional.
namespace	Specifies the namespace that contains the attributes that can be used. Optional. Possible values: - ##any = Attributes from any namespace can be used. Default. - ##other = Attributes from any namespace that is not the namespace of the parent element can be used. - ##local = Attributes must come from no namespace. - ##targetNamespace = Attributes from the namespace of the parent element can be used. - List of {URI references of namespaces, ##targetNamespace, ##local} – attributes from the space-delimited list of namespaces can be used.
processContents	Specifies how XML processor is to handle validation of elements specified by this <anyElement>. Optional. Possible values: - strict = Schemas must be obtained and elements validated. Default. - lax = As strict, but if schema cannot be obtained, no error will occur. - skip = No validation.

Anonymous Types

An element declaration can be combined with a simple or complex type declaration. The simple or complex type will then be referred to as an anonymous type because it cannot be used outside of the element declaration. An example:

```
...
<element name="person">
    <complexType>
        <sequence>
            <element name="firstName" type="string"/>
            <element name="lastName" type="string"/>
            <element name="age" type="int"/>
        </sequence>
    </complexType>
</element>
...
```

Notice that the <element> tag does not have a *type* attribute and the <complexType> tag does not have a *name* attribute.
Anonymous types can be nested.

Import and Include

Existing XML schema definitions can be imported or included in a new XML schema, in order to reuse previous definition and/or to split XML schema definitions into multiple files.

- Import
  Import a XML schema that has a different namespace from the XML schema into which it is imported.
- Include
  Include a XML schema that has the same namespace that the XML schema into which it is included.

The picture below shows the XML schema in the file “policeCaptainSchema.xsd” including an XML schema from the file “policemanSchema.xsd”, being in the same namespace, and importing an XML schema from the file “personSchema.xsd” that is in a different namespace than the police-captain schema.

1.2 XML Schema in Java EE Web Services

Describe the use of XML schema in Java EE Web services

In general, an XML schema describes a markup language; how elements and attributes are organized and what their data types are.

- An XML schema can be used to validate an XML document, in order to ensure its correctness, both in regard to structure and content.
- XML schemas/markup languages may also be reused within other XML schemas.
  For instance an XML schema describing an address entry may be reused in a XML schema describing an employee in a company (with probably has an address) and an XML schema describing an invoice (which probably is to be sent to an address) etc.
- XML schemas with different namespaces can also be used as a versioning system.
  If an XML schema changes, then the old version can be assigned one namespace and the new version can be assigned a different namespace, thus allowing the use of both the old and new version at the same time.

2. SOAP 1.2 Web Service Standards

References:
http://www.w3.org/TR/2007/REC-soap12-part0-20070427/
http://www.w3.org/TR/2007/REC-soap12-part1-20070427/
http://www.w3.org/TR/2007/REC-soap12-part2-20070427/
http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#messaging

2.1 SOAP Message Encoding Types

List and describe the encoding types used in a SOAP message.

The following encoding types may be used in SOAP messages:

- SOAP encoding.
- Literal encoding.

SOAP encoding:

- Describes how to map non-XML based data to a format that can be transferred in SOAP messages.
- Is less flexible and more restricted compared to XML.
- Is optional and does not have to be implemented in a SOAP node.
- Is not allowed by the WS-I Basic Profile.
  The WS-I Basic Profile prohibits use of all encodings, since this has been a cause of interoperability problems.

For details on SOAP encoding, please refer to the SOAP 1.2 specification documents and this webpage: http://www.herongyang.com/Web-Services/SOAP-Encoding-What-Is-It.html

“Literal” means that the XML document fragment can be validated against its XML schema.
Messaging modes using the literal encoding are:

- Document/Literal
- RPC/Literal

SOAP messaging modes describe the contents of a SOAP message.

The WS-I Basic Profile 1.1 defines the bindings as follows:

RPC/Literal Binding

An "rpc-literal binding" is a wsdl:binding element whose child wsdl:operation elements are all rpc-literal operations.

An "rpc-literal operation" is a wsdl:operation child element of wsdl:binding whose soapbind:body descendant elements specify the use attribute with the value "literal", and either:

1. The style attribute with the value "rpc" is specified on the child soapbind:operation element; or
2. The style attribute is not present on the child soapbind:operation element, and the soapbind:binding element in the enclosing wsdl:binding specifies the style attribute with the value "rpc".

Document/Literal Binding

A "document-literal binding" is a wsdl:binding element whose child wsdl:operation elements are all document-literal operations.

A "document-literal operation" is a wsdl:operation child element of wsdl:binding whose soapbind:body descendent elements specifies the use attribute with the value "literal" and, either:

1. The style attribute with the value "document" is specified on the child soapbind:operation element; or
2. The style attribute is not present on the child soapbind:operation element, and the soapbind:binding element in the enclosing wsdl:binding specifies the style attribute with the value "document"; or
3. The style attribute is not present on both the child soapbind:operation element and the soapbind:binding element in the enclosing wsdl:binding.

Document/Literal

In the Document/Literal SOAP messaging mode, the Body element of a SOAP message contains an XML document fragment; a well-formed XML element that contains arbitrary data that belongs to an XML schema and namespace separate from the SOAP message's XML schema and namespace.

An example of a Document/Literal SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:po=”http://www.books.com/purchase”>
    <soap:Header>
        <!-- Header blocks go here -->
    </soap:Header>
    <soap:Body>
        <po:purchaseOrder orderDate="2008-09-22"
            xmlns:po="http://www.somedomain.com/xyz/PO">
            <po:accountName>BookLover</po:accountName>
            <po:accountNumber>923</po:accountNumber>
            ...
            <po:book>
                <po:title>Air Guitars In Action</po:title>
                <po:quantity>300</po:quantity>
                <po:wholesale-price>14.99</po:wholesale-price>
            </po:book>
        </po:purchaseOrder>
    </soap:Body>
</soap:Envelope>
```

The Document/Literal messaging mode can use either the one-way or the request-response messaging exchange pattern

RPC/Literal

References:

http://www.w3.org/TR/2007/REC-soap12-part2-20070427/#soapforrpc

The RPC/Literal SOAP messaging mode is often used to expose traditional components as web services. Such components does not explicitly exchange XML data, but have methods with parameters and return values.
Given the following Java interface:

```
public interface BookQuote extends java.rmi.Remote
{
    // Get the wholesale price of a book
    public float getBookPrice(String ISBN)
      throws RemoteException, InvalidISBNException;
}
```

An example of RPC/Literal request and a RPC/Literal response message calling the *getBookPrice* method in the above interface may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:sd="http://www.somedomain.com/xyz/BookQuote">
   <soap:Body>
      <sd:getBookPrice>
          <isbn>0321146182</isbn>
      </sd:getBookPrice>
   </soap:Body>
</soap:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:sd="http://www.somedomain.com/xyz/BookQuote" >
   <soap:Body>
      <sd:getBookPriceResponse>
          <result>24.99</result>
      </sd:getBookPriceResponse>
   </soap:Body>
</soap:Envelope>
```

Unlike Document/Literal, with which the Body element of the SOAP message may contain any valid XML data, SOAP defines a standard XML format for RPC-style messaging. However, use and implementation of the SOAP RPC standard is optional.
The RPC/Literal messaging mode can use either the one-way or the request-response messaging exchange pattern, but it is more common to use the request-response one.

Messaging Exchange Patterns

There are two messaging exchange patters, which describe the flow of messages:

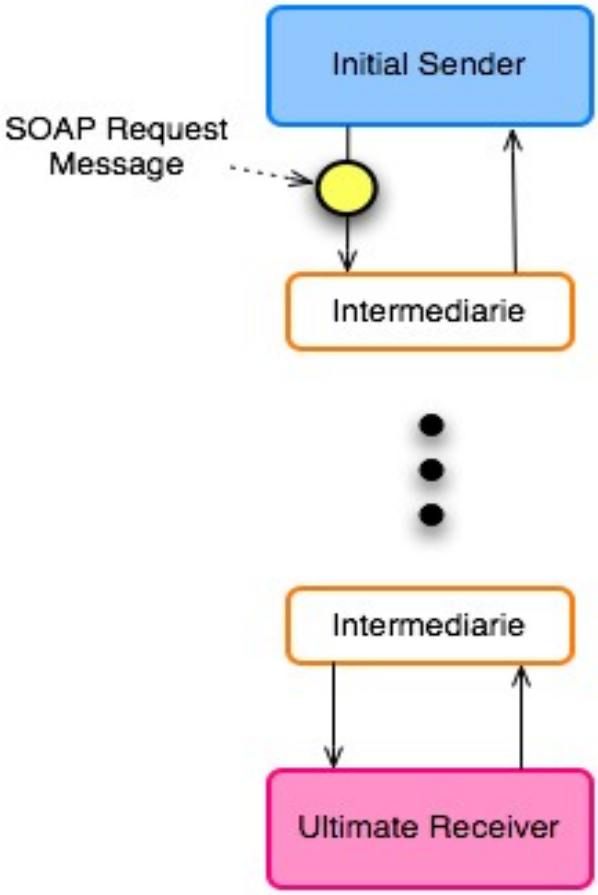
- One-way
- Request-response

2.2 SOAP Processing and Extensibility Model

Describe the SOAP Processing and Extensibility Model.

SOAP Processing Model

Reference: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/#msgexchngmdl



A SOAP message is sent by an Initial Sender, passing through zero or more Intermediaries, before reaching the Ultimate Receiver. Depending on the message exchange pattern, the Ultimate receiver may generate a SOAP response message that is returned to the Initial Sender.

SOAP Nodes and SOAP Roles

The Initial Sender, Ultimate Receiver and Intermediaries are called SOAP nodes. A SOAP node is identified by an URI. Each SOAP node can take on one or more SOAP roles, each of which is identified by a URI known as the SOAP role name. Even though role names are in the form of an URI, there are no routing or message exchange semantics associated with the SOAP role name. There are three SOAP roles defined by the SOAP 1.2 specification:

Role Name	Role URI	Description
next	http://www.w3.org/2003/05/soap-envelope/role/next	Each SOAP intermediary and the ultimate SOAP receiver MUST act in this role.
none	http://www.w3.org/2003/05/soap-envelope/role/none	SOAP nodes MUST NOT act in this role. Header blocks with this role will never be formally processed.
ultimateReceiver	http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver	The ultimate receiver MUST act in this role.

Additional roles may be specified by applications.

The role Attribute

A SOAP header block is said to be targeted at a SOAP node if the SOAP *role* attribute of the header block contains a name of a role in which the SOAP node operates.

> **Note!**
> In SOAP 1.1, the name of the attribute in header blocks indicating which role it targeted at is *actor*.

The following is an example of what a SOAP header block with an *role* attribute may look like:

```
…
<soap:Header>
    <!--
        This header block has an role attribute telling which role should process it.
    -->
    <pb:processed-by soap:role="http://www.ivan.com/processedByRole">
        <pb:node>
            <pb:time>23424872</pb:time>
            <pb:id>http://www.ivan.com/somenode</pb:id>
        </pb:node>
    </pb:processed-by>
    <!--
        Note no role attribute present, so this header block is to be processed by
        the ultimate receiver of the message and no other node.
        This is equivalent to specifying the following role URI:
        http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver
    -->
    <sec:security>
        <sec:encryption-key>a6e7f6bc9873ed6bc</sec:encryption-key>
    </sec:security>
    <!--
        This header block has an role attribute saying that the next
        node in the message chain should process it.
    -->
    <ns:myHeaderBlock soap:role=”http://www.w3.org/2003/05/soap-envelope/role/next”>
        ...
    </ns:myHeaderBlock>
</soap:Header>
...
```

The default value of the *role* attribute is http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver.
The *role* attribute is part of the same namespace as the <Envelope>, <Body> and <Header> elements of the SOAP schema.
A node determines if and how a header block is to be processed in the following way:

- The *role* attribute value, an URI, is examined to determine if the node can process the header block or not.
- If the node can process the header block, the *role* attribute content and the namespace of the header block is used to determine which code module of the node that is to process the header block. If the node cannot process the header block, it is ignored.

The mustUnderstand Attribute

The *mustUnderstande* attribute is part of the same namespace as the Envelope, Body and Header elements of the SOAP schema. This attribute determines whether the processing of a header block with an *role* attribute matching the node node role is mandatory or not.
The *mustUnderstand* attribute should only appear on the root element of a SOAP header block. If it appears elsewhere, it should be ignored.
The BP stipulates that the *mustUnderstand* attribute may only have the value “1” or “0”. The default value is “0”.

Example: If the *role* attribute of a header block is set to the standard value specifying that the next node in the message path is to process the header block and the *mustUnderstand* attribute is set to “1”, then the next node must process the header block or generate a SOAP fault.

If the *mustUnderstand* attribute is set to “0” and a node operates in the role declared in the *role* attribute, the node is only required to remove the header block. The message may then be passed on regardless of whether the node is able to process it or not (for instance, if processing fails because the node cannot recognize the XML structure or the namespace of the header block).

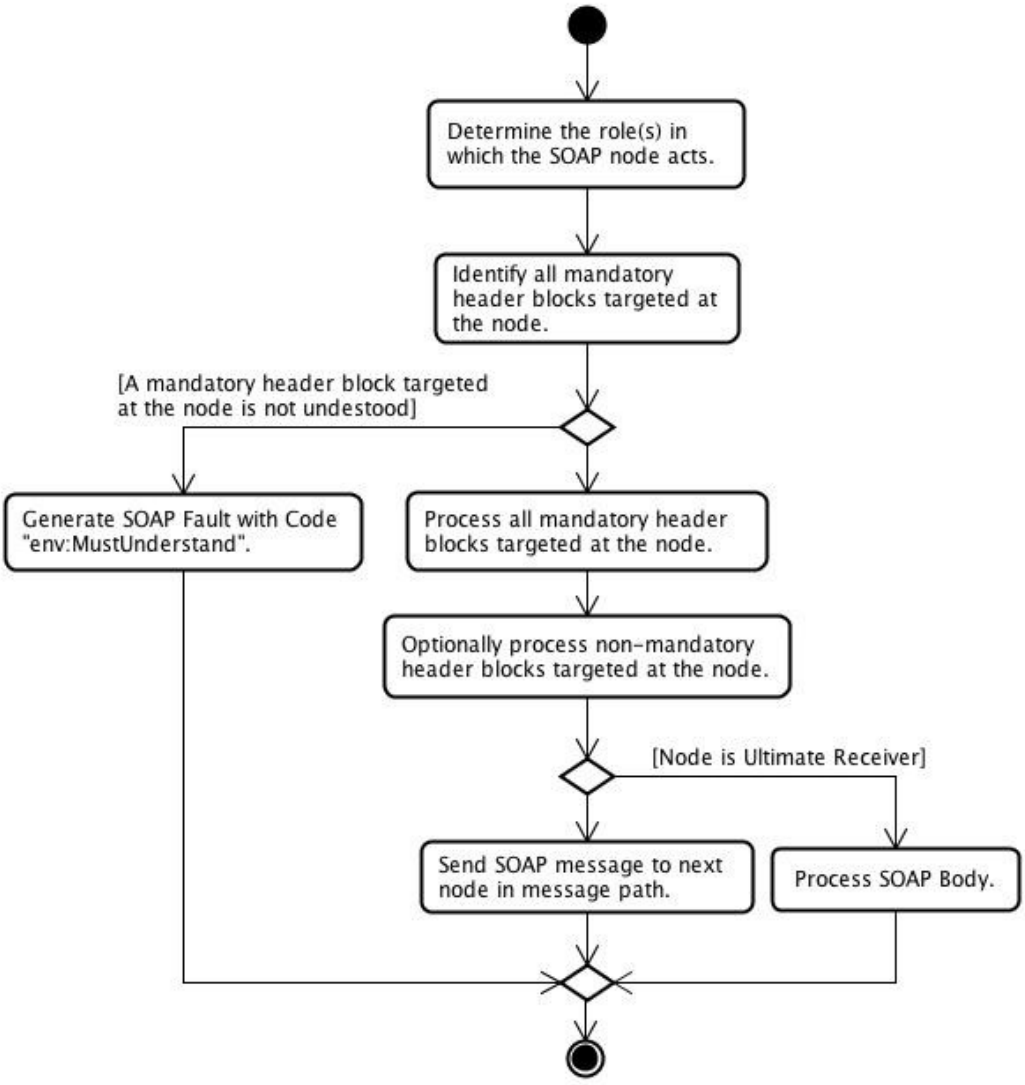
It is not considered an error if the Ultimate Receiver receives a SOAP header block with the *mustUnderstand* attribute set to “1” and the *role* attribute specifying some role that the ultimate receiver does not take on.

Processing SOAP Messages

This section describes how a SOAP node processes a single SOAP message.
The SOAP message processing model does not maintain any state, each message is processed in isolation.
A SOAP message is processed by a SOAP node using these steps:

- Determine the role(s) in which the SOAP node acts.
  The node may inspect the contents of the SOAP envelope in this step.
- Identify all header blocks targeted at the node with the *mustUnderstand* attribute set to true (mandatory header blocks).
- If one or more of the header blocks from the previous step are not understood by the SOAP node, then generate a SOAP fault with the Code set to “env:MustUnderstand”.
- Process all mandatory header blocks targeted at the node.
  The node may also choose to process header blocks that are not mandatory.
- If the node is the Ultimate Receiver, then process the SOAP body.
- If the node is an intermediary and no fault occurred, send the SOAP message along the message part.

The above expressed in an activity diagram (note that SOAP faults may also occur when processing header blocks and the SOAP body):



Further notes about the processing of SOAP messages:

- The processing of one SOAP message may result in at most one SOAP fault.
- A SOAP node processing a header block or the SOAP body may reference any information in the SOAP envelope.
- SOAP header blocks may be processed in arbitrary order.

Relaying SOAP Messages

There are two kinds of SOAP intermediaries; forwarding intermediaries and active intermediaries.

SOAP Forwarding Intermediaries

A SOAP forwarding intermediary must process a SOAP message, as described above, and additionally:

- Remove all processed SOAP header blocks.
- Remove all non-relayable header blocks that were targeted at the node but ignored during processing.
- Retain all relayable header blocks that were targeted at the node but ignored during processing.

Relayable Header Blocks

A SOAP header block may have a *relay* attribute that indicates whether the header block is relayable or not. Example of header blocks with and without the *relay* attribute:

```
<env:Envelope
    xmlns:env="http://www.w3.org/2003/05/soap-envelope"
    xmlns:xml="http://www.w3.org/XML/1998/namespace">

    <env:Header>
        <test:echoOk xmlns:test="http://example.org/ts-tests"
            env:role="http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
            env:mustUnderstand="1" env:relay="false"
            env:anyAttribute="any value">
            foo
        </test:echoOk>
        <test1:echoOk1 xmlns:test1="http://example1.org/ts-tests"
            env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
            env:mustUnderstand="0" env:relay="true"
            env:anyAttribute="any value">
            foo
        </test1:echoOk1>
        <test2:echoOk2 xmlns:test2="http://example2.org/ts-tests"
            env:role="http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver"
            env:mustUnderstand="1"
            env:anyAttribute="any value">
            foo
        </test2:echoOk2>
    </env:Header>
    ...
```

The default value of the *relay* attribute is false. The *relay* attribute should only appear on the root element of a SOAP header block. If it appears elsewhere, it may be ignored.
The Basic Profile does not say anything about values of the relay attribute, since it is new for SOAP 1.2 and the Basic Profile was written when the latest SOAP version was 1.1.

The forwarding behaviour of a SOAP node is described in the following table:

Role		Header Block	
Short Name	**SOAP Node is in Role**	**Understood & Processed**	**Forwarded**
next	Yes	Yes	No, unless reinserted
		No	According to *relay* attribute
user-defined	Yes	Yes	No, unless reinserted
		No	According to *relay* attribute
	No	N/A	Yes
ultimateReceiver	Yes	Yes	N/A
		No	N/A
none	No	N/A	Yes

The following rules apply to the preservation of the XML data in a SOAP message by a SOAP forwarding intermediary:

1. The entire SOAP message must be preserved, except as specified in the following rules.
2. SOAP header blocks may be removed.
3. SOAP header blocks may be added.
4. White space characters may be added to or removed from:
   - Children of the SOAP Envelope element.
   - Children of the SOAP Header element.
5. Comment elements may be added to or removed from:
   - Children of the SOAP Envelope element.
   - Children of the SOAP Header element.
6. Attribute items may be added to:
   - The SOAP Envelope.
   - The SOAP Header.
7. Namespace attributes may be added to:
   - The SOAP Envelope.
   - The SOAP Header.
8. If the *role* attribute of a SOAP header block is Ultimate Receiver, then it may be omitted.
9. The *mustUnderstand* attribute of a SOAP header block can be transformed:
   - The value “false” may be substituted with “0”.
   - The value “true” may be substituted with “1”.
   - The attribute may be omitted if its value is “false”.
10. The *relay* attribute of a SOAP header block can be transformed:
    - The value “false” may be substituted with “0”.
    - The value “true” may be substituted with “1”.
    - The attribute may be omitted if its value is “false”.
11. The base URI of the document information item need not be maintained.
    See http://www.w3.org/TR/xml-infoset/#infoitem.document for definition of what the document information item is.
12. The base URI of element information items in the SOAP message may be changed or removed.
13. The character encoding property of the document information item may be changed or removed.
14. All namespace information items in elements must be preserved.
15. Additional namespace information items may be added.

The rules above allow for signing of SOAP header blocks, the SOAP body, and combinations of SOAP header blocks and the SOAP body.

SOAP Active Intermediaries

In addition to the processing of SOAP messages performed by forwarding intermediaries, active intermediaries may also process SOAP messages in ways not described by SOAP header block(s) found in the incoming SOAP message. Example of services offered by SOAP active intermediaries are: Security services, annotation services, and content manipulation services.

SOAP Versioning Model

The version of a SOAP 1.2 message is specified by the local name of the SOAP Envelope (which is "Envelope") and the namespace name of the SOAP Envelope.
SOAP 1.2 has the namespace name http://www.w3.org/2003/05/soap-envelope.
Versioning only concerns the SOAP Envelope element, it does not address versioning of SOAP header blocks, encodings, protocol bindings, or anything else.

SOAP Extensibility Model

Reference: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/#extensibility

SOAP Features

A SOAP feature is an extension of the SOAP messaging framework. Some examples of SOAP features are "reliability", "security", "correlation", "routing", and message exchange patterns such as request/response, one-way, and peer-to-peer conversations.
The SOAP extensibility model provides two ways to express SOAP features:

- The SOAP processing model
  Describes the behaviour of a single SOAP node in connection to processing an individual SOAP message.
- The SOAP protocol binding framework
  Mediates the act of sending and receiving SOAP messages by a SOAP node via an underlying protocol.

The SOAP processing model makes it possible for SOAP nodes that include mechanisms to implement one or more features to express such features within the SOAP envelope as SOAP header blocks. Header blocks can be targeted at one or more SOAP nodes in a SOAP message path. The combined syntax and semantics of SOAP header blocks are called a SOAP module.

A SOAP protocol binding operates between two adjacent nodes in a SOAP message path. The WS-I Basic Profile only allows for SOAP over HTTP, so the SOAP protocol binding framework will not be further explored.
The specification of a SOAP feature must include the following:

- A name in the form of an URI.
- The information (state) required at each node to implement the feature.
- The processing required at each node to fulfill the obligations of the feature.
  This includes handling of errors in underlying protocol.
- The information that is to be transferred from node to node.

SOAP Message Exchange Patterns

A SOAP message exchange pattern is a template describing the pattern by which messages are exchanged between SOAP nodes.
The specification of a message exchange pattern must include the following:

- A name in the form of an URI.
- A description of the life-cycle of a message exchange adhering to the pattern.
- A description of temporary relationships between SOAP nodes, if any, that are formed when multiple messages are exchanged.
  For instance: In the Request-Response message exchange pattern, the Ultimate Receiver is to send a response message back to the Initial Sender.
- Descriptions of successful and unsuccessful termination of a message exchange using the pattern.

A message exchange pattern is, as before, a SOAP feature and the specification of the pattern must include the same things as required for a SOAP feature (four first items repeated for convenience) and additional items (last two items):

- A name in the form of an URI.
- The information (state) required at each node to implement the feature.
- The processing required at each node to fulfill the obligations of the feature.
  This includes handling of errors in underlying protocol.
- The information that is to be transferred from node to node.
- Any requirements concerning additional messages to be generated.
  For instance, response messages in the Request-Response pattern.
- Rules for how to deliver, or otherwise process, SOAP faults that may occur during a message exchange.

SOAP Modules

A SOAP module is the specification of the syntax and semantics of one or more SOAP header blocks. A SOAP module specification must follow these rules:

- Must provide an URI used as an identifier for the module.
- Must declared the features provided by the module.
- Must specify the contents and semantics of the header blocks used to implement the behaviour in question.
- Must specify any known interactions with, or changes to the interpretation of, the SOAP body.
  For example, we can imagine a module which encrypts and removes the SOAP body, inserting instead a SOAP header block containing a checksum and an indication of the encryption mechanism used. The specification for such a module would indicate that the decryption algorithm on the receiving side is to be run prior to any other modules which rely on the contents of the SOAP body.

2.3 SOAP Message Construct and SOAP Messages with Attachments

Describe SOAP Message Construct and create a SOAP message that contains an attachment.

Reference: http://www.w3.org/TR/soap12-part1/#soapenv

SOAP is a protocol that allows the exchange of structured data specified by an XML schema between peers in a decentralized, distributed environment.

SOAP Message Elements

The basic structure of a SOAP message looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

      <soap:Header>
            <!-- Data contained in the SOAP header inserted here. -->
      </soap:Header>

      <soap:Body>
            <!-- Data contained in the SOAP envelope inserted here. -->
      </soap:Body>

</soap:Envelope>
```

A SOAP message consists of three elements; the envelope, the header and the body elements, in the order as in the above example.

Envelope Element

Reference: http://www.w3.org/TR/soap12-part1/#soapenvelope

- Belongs to the http://www.w3.org/2003/05/soap-envelope namespace.
- Root of a SOAP message.
- Name of the element is <Envelope>.
- Contains one optional <Header> element.
- Contains one <Body> element.

Header Element

Reference: http://www.w3.org/TR/soap12-part1/#soaphead

- Belongs to the http://www.w3.org/2003/05/soap-envelope namespace.
- Name of the element is <Header>.
- <Header> element of SOAP message is optional.
- Contain zero or more distinct XML elements, called header blocks.
- Examples of types of contents are: Security credentials, transaction ids, routing instructions, debugging information etc.

Header blocks have the following properties:

- Each header block must have its own namespace.
- Contains zero or more XML elements.

The header block root element may have zero or more of the following attributes:

Attribute Name	Description
encodingStyle	Specifies the encoding of the header block. See below!
role	Specifies the SOAP node(s) at which the header block is targeted. See above! Default value is http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver.
mustUnderstand	Specifies whether processing of the header block is mandatory or optional. See above!
relay	Indicates whether the header block is relayable or not. See above! Default value is false.

The *role*, *mustUnderstand* and *relay* attributes should only be used on the root element of a SOAP header block. If used elsewhere, they should be ignored.

Example of a SOAP header with a single SOAP header block:

```
...
<env:Header xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
    <t:Transaction
        xmlns:t="http://example.org/2001/06/tx"
        env:mustUnderstand="true">
        5
    </t:Transaction>
</env:Header>
...
```

Body Element

Reference: http://www.w3.org/TR/soap12-part1/#soapbody

- Belongs to the http://www.w3.org/2003/05/soap-envelope namespace.
- The name of the element is <Body>.
- A SOAP message must contain exactly one body element.
- The body element may contain any well-formed XML data or be empty.
- The body element contains either application data or a SOAP fault message.
- The body element must be an immediate child of the <Envelope> element.
- The body element must follow the <Header> element, if any, or be the first child of the <Envelope> element.

Body Element Children

All child elements of the SOAP Body element:

- Should have a namespace specified.
- May have zero or more children, which in turn may be namespace qualified.
- May have zero or more attributes specified.
  For instance the *encodingStyle* attribute (restrictions apply, see next section).

The SOAP encodingStyle Attribute

Reference: http://www.w3.org/TR/soap12-part1/#soapencattr

The SOAP *encodingStyle* attribute is commonly used to specify the encoding used with SOAP RPC messages. It belongs to the "http://www.w3.org/2003/05/soap-envelope" namespace and its value can be any URI that is understood by the sender and receiver of the message.

Example of usage:

```
<soap:Envelope
 xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ik="http://www.ivan.com/calculator">
   <soap:Body>
      <ik:add soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
        <num1 xsi:type="int">1</num1>
        <num2 xsi:type="int">2</num2>
      </ik:add>
   </soap:Body>
</soap:Envelope>
```

The *encodingStyle* attribute may, according to the SOAP 1.2 specification, only be used in the following places in a SOAP message:

- Anywhere in a SOAP header block, including the root element of the header block.
- Anywhere in the SOAP body element except on SOAP fault elements and the SOAP body element itself.
- Anywhere in a SOAP detail element, excluding the SOAP detail element itself.

The scope of an *encodingStyle* declaration is that of the element on which it is declared and all its descendants, excluding any elements which have an *encodingStyle* specified and their descendants. The default encoding style is http://www.w3.org/2003/05/soap-envelope/encoding/none, which means that no claims are made regarding encoding.

SOAP Faults

Reference: http://www.w3.org/TR/soap12-part1/#soapfault

A SOAP message that contains a <Fault> element in its <Body> element is called a fault message. Fault messages are used to report errors to nodes earlier in the message path. Possible reasons for faults may be improper message formatting, version mismatches, trouble processing a header and application-specific errors.
Example of a SOAP fault message:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:m="http://www.example.org/timeouts"
              xmlns:xml="http://www.w3.org/XML/1998/namespace">
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:Sender</env:Value>
                <env:Subcode>
                    <env:Value>m:MessageTimeout</env:Value>
                </env:Subcode>
            </env:Code>
            <env:Reason>
                <env:Text xml:lang="en">Sender Timeout</env:Text>
                <env:Text xml:lang="sv">Avsändar timeout</env:Text>
            </env:Reason>
            <env:Detail>
                <m:MaxTime>P5M</m:MaxTime>
            </env:Detail>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

If an error occurs, a SOAP node follows the following procedure to process the error:

- A SOAP fault is generated by the node, be it an intermediary or the ultimate receiver.
- If the messaging exchange pattern One-Way is used, then the SOAP fault must not be transmitted to the immediate sender. The SOAP fault may be stored somewhere.
- If the messaging exchange pattern Request-Response is used, then the SOAP fault must be transmitted to the immediate sender, that is, the SOAP node immediately preceding the current SOAP node in the messaging chain.
  The node receiving the SOAP fault may, in turn, take some action (such as undoing operations) and also, optionally, transmit the SOAP fault further up the SOAP messaging path to its previous node, if any.

If the SOAP Body element contains a <Fault> element, it must contain only one single <Fault> element and nothing else.

A SOAP <Fault> element has the following contents:

Element Name	Required	Description
Code	Yes	Identifies the error. See below!
Reason	Yes	Human-readable description of the error in one or more <Text> items. See below!
Node	If the node causing the fault is not the Ultimate Receiver.	URI of the node in the message path causing the fault.
Role	No	The role, an URI, in which the node at which the fault occurred was operating in when the fault occurred.
Detail	No	Zero or more XML fragments holding application specific error information.

The <Code> Element

Reference: http://www.w3.org/TR/soap12-part1/#faultcodeelement
The SOAP <Code> element may contain the following child elements:

Element Name	Required	Description
Value	Yes	High-level description of the fault, using a set of predefined SOAP fault codes. See below!
Subcode	No	Contains <Value> element further detailing the description of the fault. May contain application-defined subcodes.

Example of a <Code> element inside a SOAP <Fault> element:

```
...
    <env:Code>
        <env:Value>env:Sender</env:Value>
        <env:Subcode>
            <env:Value>m:MessageTimeout</env:Value>
        </env:Subcode>
    </env:Code>
...
```

Note that the value of the <Value> element belongs to the SOAP envelope namespace while the value of the <Value> element in the <Subcode> element does not belong to the SOAP envelope namespace.

The following table lists the possible values of the <Value> element in the <Code> element of a SOAP fault:

Faultcode	Description	Possible Causes	Fault Receiver Action
Sender	SOAP message sent to the node caused the error. The node cannot process the message because there is something wrong with the message or its data.	SOAP message not well-formed, contains invalid data or lacks expected information.	Do not resend same message, but try to correct the problem or abort completely.
Receiver	Node receiving the SOAP message malfunctioned or otherwise unable to process the message.	Unavailability of some resource required when processing the message. Node has encountered an abnormal condition.	Message can be assumed to be correct and resent after some time.
VersionMismatch	Receiving node does not recognize the namespace of the SOAP message's Envelope, Header, Body or Fault elements. Root element of the message is not an Envelope element.	SOAP message is properly structured, but it uses elements and namespaces in the Body element that the receiver doesn't recognize. SOAP message specifies a namespace for the SOAP <Envelope> element and its children that is not the SOAP namespace of appropriate version.	Do not resend same message, but try to correct the problem or abort completely.
MustUnderstand	The node does not understand a header block that is targeted at the node (using the *role* attribute) and that have the *mustUnderstand* attribute set to 1.	SOAP message contains a mandatory header block that the receiver doesn't recognize.	
DataEncodingUnknown	A SOAP header block or SOAP body child element has a data encoding that the faulting node does not understand. See section on the *encodingStyle* attribute!	See description!	Do not resend same message, but try to correct the problem or abort completely.

VersionMismatch Faults

Reference: http://www.w3.org/TR/soap12-part1/#faultcodes

If a SOAP node generates a fault with the value of the <Value> element in the <Code> element being “env:VersionMismatch”, a SOAP <Upgrade> header block should be included in the fault message. The upgrade header block lists the fully qualified names of SOAP envelopes the node understands using <SupportedEnvelope> elements. The ordering of the <SupportedEnvelope> elements is significant; the most preferred version is listed first and the least preferred version last etc.
A <SupportedEnvelope> element uses the *qname* attribute to specify the fully qualified name of an envelope element which the node understands.
The *encodingStyle* attribute must never be used in the <Upgrade> element.

Example of SOAP <Fault> with <Upgrade> header block:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
              xmlns:xml="http://www.w3.org/XML/1998/namespace">
    <env:Header>
        <env:Upgrade>
            <env:SupportedEnvelope qname="ns1:Envelope"
                xmlns:ns1="http://www.w3.org/2003/05/soap-envelope"/>
            <env:SupportedEnvelope qname="ns2:Envelope"
                xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"/>
        </env:Upgrade>
    </env:Header>
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:VersionMismatch</env:Value>
            </env:Code>
            <env:Reason>
                <env:Text xml:lang="en">Version Mismatch</env:Text>
            </env:Reason>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

MustUnderstand Faults

Reference: http://www.w3.org/TR/soap12-part1/#mufault
If a SOAP node generates a fault with the value of the <Value> element in the <Code> element being “env:MustUnderstand”, one or more SOAP <NotUnderstood> header block(s) should be included in the fault message. The node is not required to generate <NotUnderstood> header blocks for all of the header blocks that were not understood.
A <NotUndestood> element uses the *qname* attribute to specify the fully qualified name of the header block which the node did not understands.
The *encodingStyle* attribute must never be used in the <NotUnderstood> element.

Assume the following SOAP message with two header blocks that will not be understood by the Ultimate Receiver:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope'>
    <env:Header>
        <!-- First header block that will not be understood. -->
        <abc:Extension1 xmlns:abc='http://example.org/2001/06/ext'
            env:mustUnderstand='true/>

        <!-- Second header block that will not be understood. -->
        <def:Extension2 xmlns:def='http://example.com/stuff' env:mustUnderstand='true'/>
    </env:Header>
    <env:Body>
    . . .
    </env:Body>
</env:Envelope>
```

Note the root elements of the header blocks; <Extension1> and <Extension2> respectively.

The Ultimate Receiver should, in response to the above message, generate the following SOAP fault message:

```
<?xml version="1.0" ?>
<env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope'
              xmlns:xml='http://www.w3.org/XML/1998/namespace'>
    <env:Header>
        <env:NotUnderstood qname='abc:Extension1'
            xmlns:abc='http://example.org/2001/06/ext'/>
        <env:NotUnderstood qname='def:Extension2'
            xmlns:def='http://example.com/stuff'/>
    </env:Header>
    <env:Body>
        <env:Fault>
            <env:Code>
                <env:Value>env:MustUnderstand</env:Value>
            </env:Code>
            <env:Reason>
                <env:Text xml:lang='en'>
                    One or more mandatory SOAP header blocks not understood
                </env:Text>
            </env:Reason>
        </env:Fault>
    </env:Body>
</env:Envelope>
```

The <Reason> Element

The <Reason> element contains one or more <Text> elements, each containing a human-readable description of the error in a specified language. All <Text> elements must have the *xml:lang* attribute specified and if there are multiple <Text> elements, then each element's *xml:lang* attribute should have a different value.

```
...
    <env:Reason>
        <env:Text xml:lang="en">Sender Timeout</env:Text>
        <env:Text xml:lang="sv">Avsändar timeout</env:Text>
    </env:Reason>
...
```

The <Node> Element

The <Node> element indicates which node generated the fault (the faulting node). Including the <Node> element is required if the faulting node is an intermediary, but is optional if the faulting node is the ultimate receiver.
The <Node> element can contain any URI, but usually contains one of the following:

- The internet address of the node.
- The URI matching the *role* attribute of the node.

The <Role> Element

The optional <Role> element contains an URI specifying which role the faulting node was operating in when the fault occurred.

The <Detail> Element

The <Detail> element contains an XML fragment providing further detail about the fault. As opposed to in SOAP 1.1, the presence or absence of the <Detail> element has no relation to which part of a SOAP message was processed when the fault occurred.
The <Detail> element may contain any number of application-specific elements, detail entries. The <Detail> element itself may contain any number of qualified attributes, as long as they do not belong to the SOAP 1.2 namespace.
Example of a <Detail> element:

```
...
    <env:Detail>
        <m:MaxTime>P5M</m:MaxTime>
    </env:Detail>
...
```

Detail Entries

Each detail entry:

- May be namespace qualified.
- May have any number of child elements (detail entries).
- May have zero or more attributes.

One of the possible attributes a detail entry may have is the SOAP *encodingStyle* attribute, which specifies the encoding of the detail entry. See above!

SOAP Messages with Attachments (SwA)

SOAP message with attachments (SwA) provides a way to attach binary data to a SOAP message. SwA is based on MIME (multipurpose internet mail extension).
The following example shows a SOAP message wrapped in a MIME envelope which in turn is wrapped in a HTTP POST request.

- The Content-Type in the HTTP POST message header is Multipart/Related.
  This allows the root MIME part, that contains the SOAP message, to refer to other MIME parts.
- The *start* parameter in the Content-Type of the HTTP POST message header specifies the root MIME part.
- The recommended way to refer to other MIME parts is by using CIDs (content ids).
  A CID is always prefixed by the string “cid:”.

When using SOAP messages without attachments, the message just contains the SOAP message, as in this example:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope">
    <SOAP-ENV:Header>
        ...
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        ...
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Naturally, the above SOAP message will be wrapped in a HTTP request when being sent using HTTP.
If we add an attachment to a SOAP message, the SOAP message will become wrapped in a MIME envelope, as seen in the following example:

```
------=_Part_1_9144903.1228438463099
Content-Type: text/xml; charset=utf-8

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope">
    <SOAP-ENV:Header />
    <SOAP-ENV:Body>
        <ik:attachmentHolder xmlns:ik="http://ivan.com/attach">
            <attachment href="cid:koentent_ajdi_0" />
        </ik:attachmentHolder>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
------=_Part_1_9144903.1228438463099
Content-Type: text/plain
Content-Id: koentent_ajdi_0

This file is to be attached to a SOAP message.
It contains some very important data that are to be sent to the web service.
Blah blah blah.

------=_Part_1_9144903.1228438463099--
```

The contents of the file attached to the above SOAP message was:

```
This file is to be attached to a SOAP message.
It contains some very important data that are to be sent to the web service.
Blah blah blah.
```

The above SOAP messages was programmatically generated using SAAJ.

WS-I Basic Profile on SOAP

References:
WS-I Basic Profile 1.1 Specification, chapter 3.
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

As far as I can see, there are no topic that deals with what the WS-I Basic Profile has to say about SOAP, but I nevertheless feel that it is an important topic, so I have included this additional section. The Basic Profile 1.1, referred to as BP hereafter, concerns itself with SOAP version 1.1.
Parts of the information in this section are already present in the section on SOAP.

SOAP Envelopes

References:

http://www.ws-i.org/Profiles/BasicProfile-1.1.html#SOAPENV

- A SOAP Envelope must conform to the structure described in the SOAP 1.1 specification, section 4:
  Reference: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383494
- The SOAP <Body> element in a SOAP <Envelope> element must have zero or one child element.
- A receiver of an envelope that does not belong to the http://schemas.xmlsoap.org/soap/envelope/ namespace (the SOAP 1.1 namespace) must generate a fault.
- Child elements of the SOAP <Body> element must be namespace qualified.
- A SOAP Envelope must not contain a Document Type Declaration (DTD).
- A SOAP Envelope must not contain Processing Instructions.
- A SOAP Envelope should not contain the namespace declaration: xmlns:xml="http://www.w3.org/XML/1998/namespace"
- The SOAP <Envelope> element must not have any child elements following the SOAP <Body> element.
- A SOAP Envelope must not contain *soap:encodingStyle* attributes on any elements belonging to the same namespace as the SOAP <Envelope> element (http://schemas.xmlsoap.org/soap/envelope/).
- A SOAP Envelope must not contain any *soap:encodingStyle* attributes on any child element of the SOAP <Body> element.
- A SOAP Envelope that has a RPC/Literal binding must not contain soap:encodingStyle attributes on any element that is a grandchild of the SOAP <Body> element.
- Any *soap:mustUnderstand* attribute appearing in a SOAP Envelope is only allowed to have the value “0” or “1”.
- A receiver must not require the use of the *xsi:type* attribute in SOAP Envelopes, except as a means to indicate a derived type (see the subsection on Polymorphism and Abstract Base Types in this section).
- The following SOAP elements must not have attributes belonging to the same namespace as the SOAP <Envelope> element (http://schemas.xmlsoap.org/soap/envelope/): <Envelope>, <Header>, <Body>.

SOAP Processing Model

References:
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#SOAPMEM
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383491

- A receiver of a SOAP message must process SOAP Envelopes so that it at least appears as if all mandatory header blocks, i.e. header blocks in which the *soap:mustUnderstand* attribute has the value "1", has been checked prior to the processing of the message.
  This in order to avoid side-effects as a result of noticing a mandatory header block after having processed (some part of) the message.
- A receiver of a SOAP message that contains a mandatory header block targeted at the receiver, using the *soap:role* attribute in the header block, must generate a "soap:MustUnderstand" fault if the receiver does not understand the header block.
- When a fault is generated, the receiver should not perform any further processing of the SOAP Envelope, except for any processing required to roll back the effects of the processing of the SOAP Envelope up to the point of the fault.
- If the normal processing of a SOAP Envelope would have resulted in a SOAP response being sent, a receiver must transmit a fault in place of the response if a fault is generated.
- A receiver that generates a fault should notify the end user, where possible, that a fault has been generated.

SOAP Faults

References:
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#SOAPFAULT

- A receiver must interpret a SOAP message as a fault when the SOAP <Body> element of the message has a single child element that is a SOAP <Fault> element.
  That is, the receiver must not rely on HTTP status codes or similar, but must always inspect the SOAP message to determine if it contains a fault or not.
- When a SOAP message is a fault, the SOAP <Fault> element must not have any other child elements than: <faultcode>, <faultactor>, <faultstring>, <detail>
- The above child elements of a SOAP <Fault> element must not be namespace qualified.
  They are local to the SOAP <Fault> element and thus does not need to be namespace qualified.
- A receiver of a fault must be able to handle faults in which the <detail> element has zero or more child elements that can be namespace qualified or unqualified.
- A receiver of a fault must be able to handle faults in which the <detail> element has zero or more attributes that can be namespace qualified or unqualified.
  The attributes must not belong to the namespace of the SOAP Envelope (http://schemas.xmlsoap.org/soap/envelope/).
- A receiver of a fault must be able to handle faults in which the <faultstring> element has a *xml:lang* attribute.
- When a SOAP Envelope contains a <faultcode> element, its contents should be:
  - One of the fault codes defined in SOAP 1.1, supplying any additional information in the <detail> element. Using these fault codes is preferred.
  - A qualified name whose namespace is controlled by the fault's specifying authority.
- When a SOAP Envelope contains a <faultcode> element, the contents of that elements should not use the dot notation to refine the meaning of the fault.
  This despite SOAP 1.1 allowing for this.

Use of SOAP in HTTP

References:
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#SOAPHTTP
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383526
http://www.ietf.org/rfc/rfc2616.txt
http://www.ietf.org/rfc/rfc2965.txt

- SOAP messages must be sent using either HTTP/1.0 or HTTP/1.1, where HTTP/1.1 is preferred.
- A HTTP request message must be sent using the HTTP POST method.
- HTTP messages must not use the HTTP Extension Framework (RFC2774).
- The value of the SOAPAction HTTP header field in a HTTP request message must be a quoted string.
- A receiver may generate a fault upon receiving a HTTP message in which the value of the SOAPAction HTTP header field is not a quoted string.
- A receiver must not rely on the value of the SOAPAction HTTP header field to correctly process the message.
- The 2xx HTTP status codes must be used to indicate the successful outcome of a HTTP request.
- The HTTP status code “200 OK” should be used to indicate that the HTTP response contains a SOAP message that is not a fault.
- The HTTP status codes “200 OK“ or “202 Accepted” should be used to indicate the successful outcome of a HTTP request which response does not contain a SOAP message.
- The HTTP status code “307 Temporary Redirect” must be used when redirecting a request to a different endpoint.
- Consumer of a web service may, but is not required to, automatically redirect a request when it encounters a “307 Temporary Redirect” HTTP status code in a response.
- The 4xx HTTP status codes must be used when there is a problem with the format of a request.
- The HTTP status code “400 Bad Request” should be used when a HTTP request is malformed.
- The HTTP status code “405 Method not Allowed” should be used when the method of a HTTP request message is not POST.
- The HTTP status code “415 Unsupported Media Type” should be used when the value of the Content-Type HTTP header field is not permitted by its WSDL description.
- The HTTP status code “500 Internal Server Error” must be used when the HTTP response contains a SOAP fault.
- A web service server may use HTTP cookies.
- A web service server using HTTP cookies should conform to RFC2965.
- A web service server should not require that its clients support cookies.
- HTTP cookies must be considered to have no meaning to the clients of a web service. I.e. clients are not allowed to interpret HTTP cookies.

3. Describing and Publishing (WSDL and UDDI)

3.1 WSDL in Web Services

Explain the use of WSDL in Web services, including a description of WSDL's basic elements, binding mechanisms and the basic WSDL operation types as limited by the WS-I Basic Profile 1.1.

References:
http://www.w3.org/TR/wsdl
http://www.ws-i.org/Profiles/BasicProfile-1.1.html

This section describes WSDL 1.1, which is the version of WSDL that the WS-I Basic Profile 1.1 concerns itself with.

Use of WSDL in Web Services

WSDL (web service description language) is an XML-based language for describing web services. A WSDL document is an XML document that adheres to the WSDL XML schema. Such a document can contain definitions of the following things related to a web service:

- Types
  Allows for custom data type definitions.
- Message
  Defining the data being sent to, and received from, the web service.
  For instance, with RPC, this means the parameter(s) and return value(s) of the procedure(s) of the web service.
- Operation
  Defines one abstract operation supplied by the web service.
- Port Type
  Defines an interface, consisting of one or more operations, of a web service.
- Binding
  Specifies the protocol and data format to be used by a particular port type.
- Port
  Defines an web service endpoint by assigning a network address to a binding.
- Service
  The grouping of one or more web service endpoint(s).

WSDL's Basic Elements

Reference:
http://www.w3.org/TR/wsdl#_service
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#description

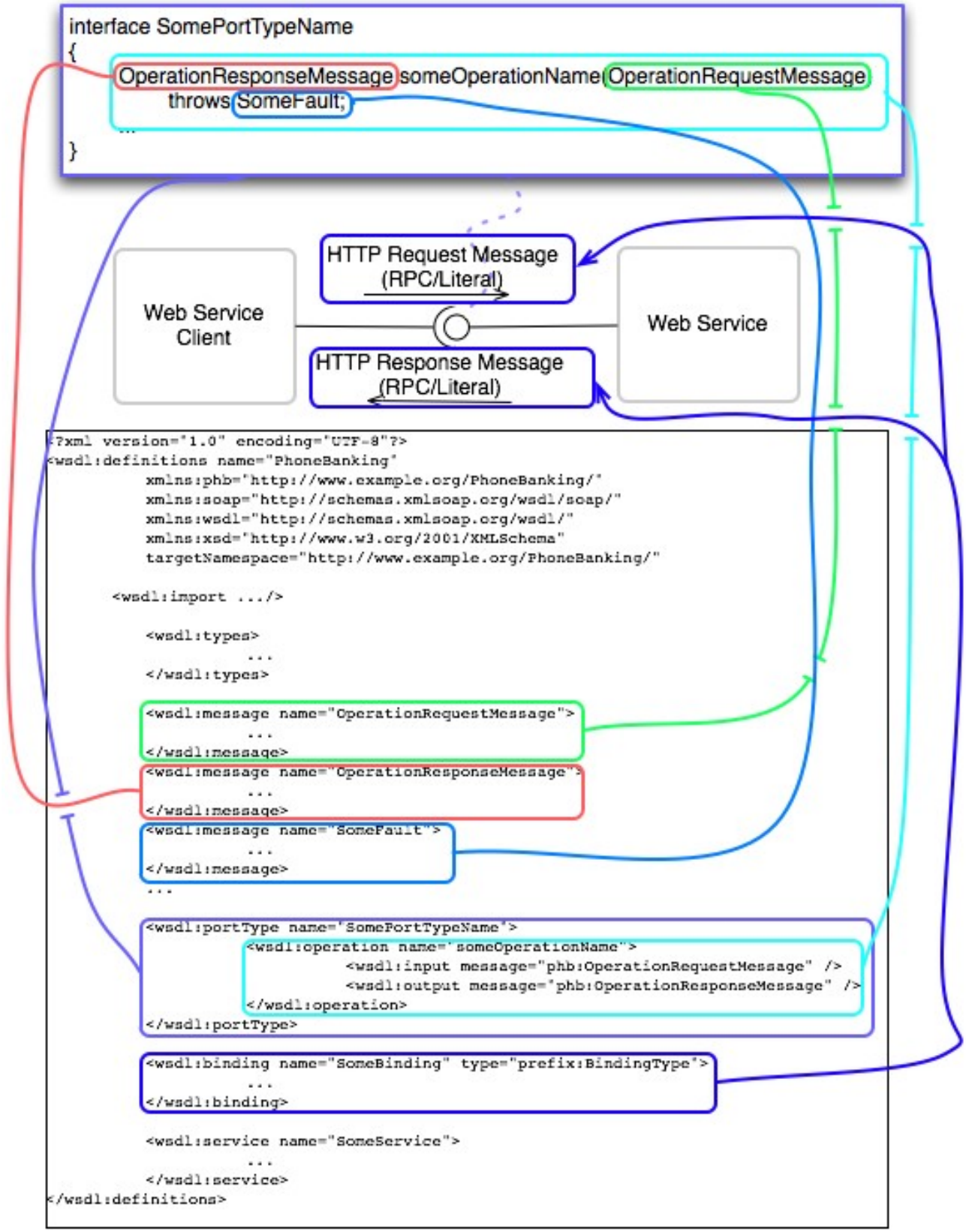
The listing below outlines the basic structure of a WSDL document – compare to the above list of what a WSDL document specifies regarding communication with a web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="PhoneBanking"
      targetNamespace="http://www.example.org/PhoneBanking/"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">

       <wsdl:import .../>

      <wsdl:types>
            ...
      </wsdl:types>

      <wsdl:message name="SomeMessageName">
            ...
      </wsdl:message>
      ...

      <wsdl:portType name="SomePortTypeName">
            <wsdl:operation name="someOperationName">
                  <wsdl:input message="prefix:OperationRequestMessage" />
                  <wsdl:output message="prefix:OperationResponseMessage" />
            </wsdl:operation>
      </wsdl:portType>

      <wsdl:binding name="SomeBinding" type="prefix:BindingType">
            ...
      </wsdl:binding>

      <wsdl:service name="SomeService">
            ...
      </wsdl:service>
</wsdl:definitions>
```

A WSDL document contains these important elements:

Element Type	Description
definitions	The root element of the WSDL document. Contains all the definitions of the document.
import	Import WSDL definitions from other WSDL documents.
types	Here, using the XML schema language, custom types are defined that are used to describe messages exchanged.
message	Describes the contents of messages using: - XML schema built-in types. - Complex types. - Types defined in the <types> element. - Types defined in external, imported, WSDL documents.
operation	Describes the abstract interface of a single operation of a web service: - Function name. Appears inside the <portType> element.
portType	With the <operation> element, describes the abstract interface of the web service: - Method names. - Input message(s) of methods. - Output message(s) of methods.
binding	Assigns protocols (ex SOAP 1.1) and encoding styles to methods and payloads of methods.
service	Assigns an internet address to a web service.

The picture below attempts to show which parts in a WSDL document corresponds to which parts in the interaction between a web service and its clients.

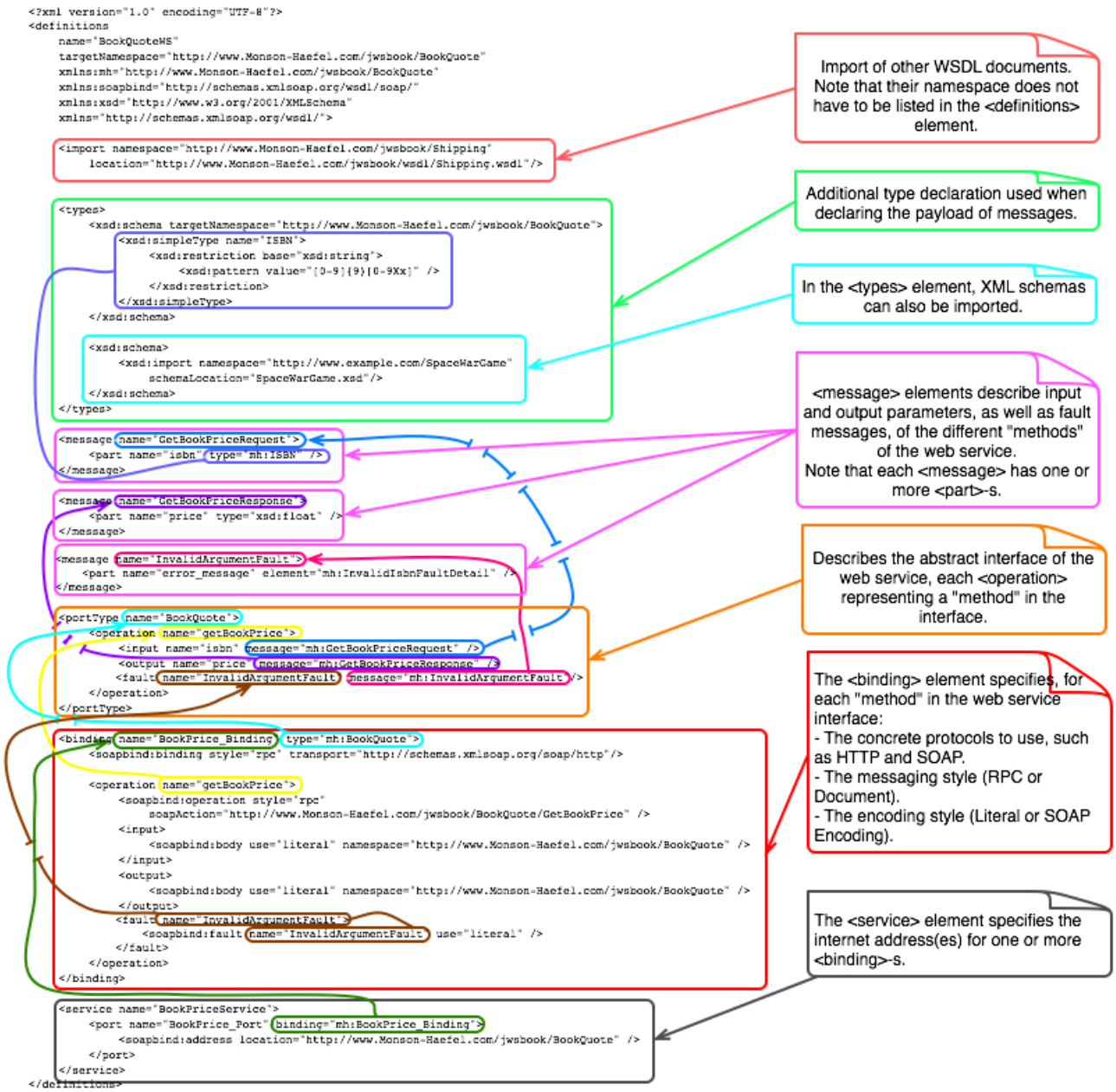


Below follows an example of what a WSDL document may look like. Note that the ordering of the child elements of the <definitions> element is significant; for instance should the <import> element appear before the <types> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    name="BookQuoteWS"
    targetNamespace="http://www.somedomain.com/xyz/BookQuote"
    xmlns:sd="http://www.somedomain.com/xyz/BookQuote"
    xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <!-- Import of other WSDL documents. This is just an example. -->
    <import namespace="http://www.somedomain.com/xyz/Shipping"
        location="http://www.somedomain.com/xyz/wsdl/Shipping.wsdl"/>

    <!--
        types element contains additional types used when declaring the parts
        (payload) of messages.
    -->
    <types>
        <xsd:schema targetNamespace="http://www.somedomain.com/xyz/BookQuote">
            <!-- The ISBN simple type -->
            <xsd:simpleType name="ISBN">
                <xsd:restriction base="xsd:string">
                    <xsd:pattern value="[0-9]{9}[0-9Xx]" />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:schema>

        <!--
            This is an example how to import another XML schema into a WSDL
            document. It must be done inside the <types> element.
        -->
        <xsd:schema>
            <xsd:import namespace="http://www.example.com/SpaceWarGame"
                schemaLocation="SpaceWarGame.xsd"/>
        </xsd:schema>
    </types>

    <!-- message elements describe the input and output parameters -->
    <!-- This is the “parameter” to the “method” that retrieves a book price. -->
    <message name="GetBookPriceRequest">
        <part name="isbn" type="sd:ISBN" />
    </message>
    <!-- This is the :return value” from the “method” that retrieves a book price. -->
    <message name="GetBookPriceResponse">
        <part name="price" type="xsd:float" />
    </message>

    <!-- portType element describes the abstract interface of a Web service -->
    <portType name="BookQuote">
        <!-- The “method” getBookPrice, its parameter and its return value. -->
        <operation name="getBookPrice">
            <input name="isbn" message="sd:GetBookPriceRequest" />
            <output name="price" message="sd:GetBookPriceResponse" />
        </operation>
    </portType>

    <!-- binding tells us which protocols and encoding styles are used -->
    <binding name="BookPrice_Binding" type="sd:BookQuote">
        <!-- RPC over HTTP. -->
        <soapbind:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

        <!-- Binding for the getBookPrice operation; RPC for both input and output. -->
        <operation name="getBookPrice">
            <soapbind:operation style="rpc"
              soapAction="http://www.somedomain.com/xyz/BookQuote/GetBookPrice"/>
            <input>
                <soapbind:body use="literal"
                   namespace="http://www.somedomain.com/xyz/BookQuote"/>
            </input>
            <output>
                <soapbind:body use="literal"
                   namespace="http://www.somedomain.com/xyz/BookQuote"/>
            </output>
        </operation>
    </binding>

    <!-- service tells us the Internet address of a Web service -->
    <service name="BookPriceService">
        <port name="BookPrice_Port" binding="sd:BookPrice_Binding">
            <soapbind:address location="http://www.somedomain.com/xyz/BookQuote"/>
        </port>
```

```
    </service>
</definitions>
```

Below follows an annotated version of the above WSDL document:



We now are going to take a closer look at the different parts of a WSDL document.

The <definitions> Element

Reference: http://www.w3.org/TR/wsdl#_service
The <definitions> element is the root element of the WSDL document.

```
<definitions
    name="BookQuoteWS"
    targetNamespace="http://www.somedomain.com/xyz/BookQuote"
    xmlns:sd="http://www.somedomain.com/xyz/BookQuote"
    xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
```

- The optional *name* attribute of the <definitions> element specifies the name of the WSDL document.
- The WSDL 1.1 XML schema namespace is http://schemas.xmlsoap.org/wsdl/
- The target namespace is http://www.somedomain.com/xyz/BookQuote and its prefix is “sd”.

The <import> Element

References:
http://www.w3.org/TR/wsdl#_document-n
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDL_and_Schema_Import
Note that the <import> elements belong to the WSDL namespace!
The WSDL <import> element has the following use:

- Import the definitions from a specified namespace in another WSDL document into the current WSDL document.
- Can be used to separate abstract definitions, such as in the <types>, <message> and <portType> elements, from concrete definitions, such as those in the <binding>, <service> and <port> elements.
- Can be used to assemble multiple WSDL documents, describing different web services, into one document describing all of the web services. See example below.

```
<definitions name="AllMhWebServices" xmlns="http://schemas.xmlsoap.org/wsdl/">

    <!-- Book quote service. -->
    <import namespace="http://www.somedomain.com/xyz/BookQuote"
        location="http://www.somedomain.com/xyz/BookPrice.wsdl" />

    <!-- Purchase order service. -->
    <import namespace="http://www.somedomain.com/xyz/po"
        location="http://www.somedomain.com/xyz/wsdl/PurchaseOrder.wsdl" />

    <!-- Shipping service. -->
    <import namespace="http://www.somedomain.com/xyz/Shipping"
        location="http://www.somedomain.com/xyz/wsdl/Shipping.wsdl" />
</definitions>
```

The WSDL <import> element must declare a *namespace* attribute, which value must match that of the namespace declared in the imported WSDL document.
The WSDL <import> element must also declare a *location* attribute, which cannot be empty or null, that must point to an actual WSDL document.
The WSDL <import> element and the <types> element can be used together. If so, the <import> element should appear before the <types> element.

Each WSDL definition type, such as <service>, <port>, <message> etc. has their own name scopes so, for instance, the name of a service and the name of a message can be the same without causing conflict.

Basic Profile on WSDL and Schema Import

- A WSDL description must only use the WSDL <import> element to import other WSDL descriptions.
  That is, the WSDL <import> element must not be used to import other kinds of XML schemas and the XML <import> element must not be used to import WSDL descriptions.
- The XML <import> element must be used to import XML schema definitions.
  Additionally, it may only be used inside an <xsd:schema> element in the WSDL <types> element.
- The *namespace* attribute in a WSDL <import> element must not be a relative URI.
- The root element of documents imported using the XML <import> element must be the <schema> element from the namespace “http://www.w3.org/2001/XMLSchema”.
- An XML schema imported by a WSDL description must use UTF-8 or UTF-16 encoding.
- A WSDL description must use UTF-8 or UTF-16 encoding.
- An XML schema imported by a WSDL description must use version 1.0 of the XML W3C Recommendation.
- The *location* attribute in a WSDL <import> element must not be empty.
- The WSDL elements <documentation>, <import> and <types> must appear in that order, if present, in a WSDL description.
- A WSDL description must not contain the namespace declaration xmlns:xml="http://www.w3.org/XML/1998/namespace".
- The value of the *targetNamespace* attribute in an imported WSDL document's <definitions> element must be the same as the value of the *namespace* attribute in the WSDL <import> element importing the WSDL document.
- A WSDL description containing WSDL extensions must not use them in a way that violates requirements of the Basic Profile.
- A WSDL description should not include extension elements with a *required* attribute set to true in any WSDL element that claims conformance to the Basic Profile.
- If during the processing of a WSDL description, a consumer encounters a WSDL extension element that has a *required* attribute with set to true that the consumer does not understand or cannot process, the consumer must fail processing.

The <types> element

References:
http://www.w3.org/TR/wsdl#_types
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLTYPES

The <types> element declares additional types that are used by message definitions when declaring the parts (payload) of the messages. In the <types> element, XML schemas containing definitions that are to be used in the WSDL document can also be imported.

```
<types>
    <xsd:schema targetNamespace="http://www.somedomain.com/xyz/BookQuote">
        <!-- The ISBN simple type -->
        <xsd:simpleType name="ISBN">
            <xsd:restriction base="xsd:string">
                <xsd:pattern value="[0-9]{9}[0-9Xx]" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:schema>

    <xsd:schema>
        <xsd:import namespace="http://www.example.com/SpaceWarGame"
            schemaLocation="SpaceWarGame.xsd"/>
    </xsd:schema>
</types>

<!-- Note how the ISBN type declared above is used. -->
<message name="GetBookPriceRequest">
    <part name="isbn" type="sd:ISBN" />
</message>
```

Note that the <schema> element containing the XML schema <import> element does not specify a *targetNamespace*. This because an <import> element cannot import another XML schema into a schema with the same namespace. Also note that the <import> element is qualified by the XML schema namespace prefix!

Basic Profile on the <types> Element

- A WSDL description must not use qualified name references to WSDL components in namespaces that haven't been imported nor defined in the WSDL document referring to the component.
- A qualified name reference to an XML schema component defined in the <types> element in a WSDL description must use one of:
  - The namespace defined in the *targetNamespace* attribute in the XML <schema> element.
  - A namespace defined in the *namespace* attribute of an XML <import> element within a XML <schema> element.
- All XML <schema> elements contained in a WSDL <types> element must have a valid, non-null, *targetNamespace* attribute unless the only child element(s) are XML <import> and/or XML <annotation> element(s).
- Declarations in a WSDL description must not extend or restrict the *soapenc:Array* type.
- Declarations in a WSDL description must not use *wsdl:arrayType* attribute in the type declaration.
- Elements in a WSDL description should not be named using the convention ArrayOfXXX.
- An type defined in a WSDL document must not use the *soapenc:arrayType* attribute.
  See example at http://www.ws-i.org/Profiles/BasicProfile-1.1.html#soapenc_Array

- The target namespace of a WSDL description and the namespace of a schema definitions in the WSDL description may be the same, since they are in separate symbol spaces.

The <message> Element

References:
http://www.w3.org/TR/wsdl#_messages

http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLMSGS

One or more <message> elements describing one, or more, of the following:

- Payload of message sent to web service.
- Payload of message returned from web service.
- Contents of SOAP header blocks.
- Contents of <detail> element of SOAP faults.

Note that there is nothing in a message definition that tells us if the message is input or output. A message can, at least theoretically, be used both for input and for output.

Each <message> has a *name* attribute, which specifies the name of the message. Such a name must be unique within the WSDL document in which the <message> appears.
<message> elements contains one or more <part> elements. A <part> element can have the following attributes:

Attribute Name	Description
name	Name of the message part. Must be unique within the <message>
element	Used with Document web services. See below! Refers to a global element in an XML schema.
type	Used with RPC web services. See below! Refers to a <simpleType> or <complexType> element in an XML schema.

Use either the *element* or the *type* attribute, never both.

RPC Style Web Service Message Definitions

Definitions of <message> elements for RPC style web services look like this:

```
<definitions name="BookPrice" ...>
  ...
  <message name="GetBulkBookPriceRequest">
    <part name="isbn" type="xsd:string"/>
    <part name="quantity" type="xsd:int"/>
  </message>
  <message name="GetBulkBookPriceResponse">
    <part name="price" type="sd:prices" />
  </message>
  ...
</definitions>
```

Both input and output messages can have zero, one or multiple parts.
Note that each <part> element inside a <message> element declares a *type* attribute.

Document Style Web Service Message Definitions

Example of a message defined to be used with a document style web service. Note that a <part> element inside a <message> element declares an *element* attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="PurchaseOrderWS"
 targetNamespace="http://www.somedomain.com/xyz/PO"
 xmlns:sd="http://www.somedomain.com/xyz/PO"
 xmlns:soapbind="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema targetNamespace="http://www.somedomain.com/xyz/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:import namespace="http://www.somedomain.com/xyz/PO"
       schemaLocation="http://www.somedomain.com/xyz/po.xsd" />
    </xsd:schema>
  </types>

  <message name="SubmitPurchaseOrderMessage">
    <part name="order" element="sd:purchaseOrder" />
  </message>
  ...
</definitions>
```

Thus, a <part> element inside a <message> may declare a *type* attribute or an *element* attribute.

- When the <message> is to be used with an RPC web service, use the *type* attribute.
- When the <message> is to be used with a document style web service, use the *element* attribute.

With document style web services, the BP mandates that each message have zero or one part.

Fault Messages

SOAP fault messages are declared in the same way as above. Such a message always have one single part, which refers to the <detail> element of the fault message.

```
<definitions name="BookQuote" ...>

  <types>
    <xsd:schema targetNamespace="http://www.somedomain.com/xyz/PO">
      <!-- Import the PurchaseOrder XML schema document -->
      <xsd:element name="InvalidIsbnFaultDetail" >
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="offending-value" type="xsd:string"/>
            <xsd:element name="conformance-rules" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>

  ...
  <message name="InvalidArgumentFault">
    <part name="error_message" element="sd:InvalidIsbnFaultDetail" />
  </message>
</definitions>
```

A SOAP fault message adhering to the above definition may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:sd="http://www.somedomain.com/xyz/BookQuote" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Sender</faultcode>
      <faultstring>
        The ISBN value contains invalid characters
      </faultstring>
      <faultactor>http://www.xyzcorp.com</faultactor>

      <detail>
        <sd:InvalidIsbnFaultDetail>
          <offending-value>19318224-D</offending-value>
          <conformance-rules>
            The first nine characters must be digits. The last
            character may be a digit or the letter 'X'. Case is
            not important.
          </conformance-rules>
        </sd:InvalidIsbnFaultDetail>
      </detail>

    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Basic Profile on the <message> Element and Message Binding

Definitions:

- An RPC/Literal binding is a WSDL <binding> element in which all <operation> elements are RPC/Literal operations.
- An RPC/Literal operation is a WSDL <operation> element, that is a child of a WSDL <binding> element, whose <soapbind:body> element has a *use* attribute with the value "literal" and either:
  - The WSDL <operation> element has a <soapbind:operation> child element with the *style* attribute having the value "rpc" or
  - The *style* attribute is not present on the <soapbind:operation> element, instead the WSDL <binding> element has a <soapbind:binding> child element with the value of the *style* attribute being "rpc".

The following section from a WSDL document shows the two places where the binding style of operations can be specified; the green section specifies the binding for all the operations in the binding and the blue section specifies the style for one single operation.

```
<binding name="BookPrice_Binding" type="sd:BookQuote">
    <!-- RPC over HTTP. -->
    <soapbind:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

    <!-- Binding for the getBookPrice operation; RPC for both input and output. -->
    <operation name="getBookPrice">
        <soapbind:operation style="rpc"
          soapAction="http://www.somedomain.com/xyz/BookQuote/GetBookPrice"/>
        <input>
            <soapbind:body use="literal"
                namespace="http://www.somedomain.com/xyz/BookQuote"/>
        </input>
        <output>
            <soapbind:body use="literal"
                namespace="http://www.somedomain.com/xyz/BookQuote" />
        </output>
    </operation>
</binding>
```

- A Document/Literal binding is a WSDL <binding> element in which all <operation> elements are Document/Literal operations.
- A Document/Literal operation is a WSDL <operation> element, that is a child of a WSDL <binding> element, whose <soapbind:body> element has a *use* attribute with the value "literal" and either:
  - The WSDL <operation> element has a <soapbind:operation> child element with the *style* attribute having the value "document" or
  - The *style* attribute is not present on the <soapbind:operation> element, instead the WSDL <binding> element has a <soapbind:binding> child element with the value of the *style* attribute being "document" or
  - Neither the <soapbind:operation> element in a WSDL <operation> element nor the <soapbind:binding> element of a WSDL <binding> element has a value for the *style* attribute specified.

When using RPC, the part accessor elements refer to children of the auto-generated wrapper element that has the same local name as the operation.
Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
        <sayHelloBack xmlns="http://hello.org/wsdl">
            <String_1 xmlns="">Ivan</String_1>
        </sayHelloBack>
    </soapenv:Body>
</soapenv:Envelope>
```

In the above example, showing a SOAP message to an RPC/Literal web service, there is one single part accessor element, namely <String_1>. The <sayHelloBack> is the wrapper element that has the same local name as the operation.

With the above definitions, we can now see what the BP has to say about the <message> element and message binding:

http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLMSGS

- A Document/Literal binding in a WSDL document can, in each of its <soapbind:body> element(s), have at most one part listed in the *parts* attribute, if the *parts* attribute is specified.
- In a WSDL document, in the WSDL <binding> element, the <soapbind:binding> element must be declared (since BP only allows for SOAP web services).
  The value of the style attribute of the <soapbind:binding> element must be either “rpc” or “document”.
  The value of the transport attribute of the <soapbind:binding> element must be declared to be HTTP (in URI).
- If a <soapbind:body> element in a Document/Literal binding in a WSDL document does not specify the *parts* attribute, then the corresponding WSDL <message> element must contain zero or one WSDL <part> element.
- A WSDL <binding> element in a WSDL document may contain <soapbind:body> element(s) that specify that the SOAP <Body> contains zero parts.
- A WSDL <message> element in a WSDL document must not contain WSDL <part> elements that has both the *type* and *element* attributes specified.
- A RPC/Literal binding in a WSDL document may, in its <soapbind:body> element(s), only refer to WSDL <part> elements that has been defined using the *type* attribute.
- WSDL <part> elements used in a RPC/Literal binding are not nillable.
- A WSDL <message> element in a WSDL document may contain WSDL <part> elements that are defined using the *element* attribute, as long as those <part> elements are not referred to from <soapbind:body> element(s) in a RPC/Literal binding.
- A Document/Literal binding in a WSDL document may, in its <soapbind:body> element(s), only refer to WSDL <part> elements that has been defined using the *element* attribute.
- WSDL <message> elements in a WSDL document may contain WSDL <part> elements that are referred to from both <soapbind:header> element(s) and <soapbind:body> element(s).

- When using RPC/Literal binding, an envelope must contain exactly one part accessor element for each of the WSDL <part> elements bound to the envelope's corresponding <soapbind:body> element.
- In a Document/Literal description where the <soapbind:body> element has a *parts* attribute with the value “” (the empty string), the corresponding envelope must have an empty SOAP <Body> element.
- In a RPC/Literal description where the <soapbind:body> element has a *parts* attribute with the value “” (the empty string), the corresponding envelope must have no part accessor elements.
- A WSDL <binding> in a WSDL document may, in its <soapbind:header>, <soapbind:headerfault> and <soapbind:fault> element(s), only refer to WSDL <part> elements that has been defined using the *element* attribute.
- A <message> containing a <part> that contains an *element* attribute must, in that attribute, refer to a global element declaration in a <schema> definition in the <types> element, be it imported or not. See examples at: http://www.ws-i.org/Profiles/BasicProfile-1.1.html#Declaration_of_part_Elements

The <portType> and <operation> Elements

References:
http://www.w3.org/TR/wsdl#_porttypes
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLPORTTYPES

The abstract interface of the web service is defined in the <portType> element.

```
<wsdl:portType name="SpaceWarGame">
	<wsdl:operation name="getResultSheet">
		<wsdl:input message="tns:getResultSheetRequest" />
		<wsdl:output message="tns:getResultSheetResponse" />
		<wsdl:fault name="fault" message="tns:getResultSheet_faultMsg"/>
	</wsdl:operation>
	<wsdl:operation name="submitCommandSheet">
		<wsdl:input message="tns:submitCommandSheetRequest"></wsdl:input>
		<wsdl:output message="tns:submitCommandSheetResponse"></wsdl:output>
	</wsdl:operation>
</wsdl:portType>
```

- A WSDL document can have one or more <portType> elements, each defining an interface of a different web service.
- Each <operation> in a <portType> corresponds to a method in the interface of the web service.
- Each <operation> defines an RPC or Document style web service method.
- Each <operation> may have at most one <input> and at most one <output> element and any number of <fault> elements.
- If an <operation> is declared with one single <input> and one single <output> element, then it uses the request-response message exchange pattern.
  Such an <operation> may additionally declare zero or more <fault> elements.

- If an <operation is declared with one single <input> and no <output> element, then it uses the one-way message exchange pattern.
  Such an operation may not declare any <fault> elements.

There are also two message exchange patterns, Solicit-Response and Notification, that WSDL does not define bindings for. These message exchange patterns are not allowed by the BP.

The *name* attribute of <input> and <output> elements within an <operation> are to be unique among all <input> and <output> elements within the <portType>.
WSDL provides a default value for the *name* attribute:

- If the one-way message exchange pattern is used, the default name of the <input> message is the name of the <operation>.
- If the request-response message exchange pattern is used, the default name of the <input> message is the name of the <operation> with “Request” appended.
  The default name of the <output> message is the name of the <operation> with “Response” appended.

All <fault> elements in an <operation> must be named with a name that is unique within the <operation>.

Parameter Ordering

References: http://www.w3.org/TR/wsdl#_parameter http://www.ws-i.org/Profiles/BasicProfile-1.1.html#parameterOrder_Attribute_Construction

In WSDL, when RPC messaging is used, the client is assumed to use procedure-call semantics and the ordering of the parameters is thus significant. To indicate which part, if any, is the return type, the *parameterOrder* attribute of the <operation> element can be used.

```
<message name="GetBulkBookPriceRequest">
  <part name="isbn" type="xsd:string"/>
  <part name="quantity" type="xsd:int"/>
</message>
<message name="GetBulkBookPriceResponse">
  <part name="prices" type="sd:prices" />
</message>
<portType name="GetBulkBookPrice" >
  <operation name="getBulkBookPrice" parameterOrder="isbn quantity">
     <input name="request" message="sd:GetBulkBookPriceRequest"/>
     <output name="prices" message="sd:GetBulkBookPriceResponse"/>
  </operation>
</portType>
```

The value of the *parameterOrder* attribute is to contain:

- All the input parts.
- The output parts that are not the return value.

In the example above, there is only one single part in the output and so it is assumed to be the return value.

The following rules apply to the list of parts that is the value of the *parameterOrder* attribute:

- The ordering of the part names reflects the ordering of the parameters in the RPC signature.
- The return value part must not be included in the list.
- If a part name appears both in the <input> and in the <output> message, the parameter is an in/out parameter.
  Note that the message part must have the same type in both messages.
- If a part name appears only in the <input> message, the parameter is an in parameter.
- If a part name appears only in the <output> message, the parameter is an out parameter.

Basic Profile on the <portType> and <operation> Elements

- The order of the elements in the SOAP <Body> element of a SOAP message must be the same as the order of the WSDL <part> elements in the WSDL <message> element(s) that are bound to the SOAP message using the <soapbind:body> element.
- A WSDL document may, but does not have to, use the *parameterOrder* attribute of the WSDL <operation> element.
- Concerning the *parameterOrder* attribute of the WSDL <operation> element in the WSDL <portType> element:
  A procedure call can have only one return type, so only a single output part may be omitted from the *parameterOrder* attribute.
- Neither the Notification nor the Solicit/Response message exchange patterns may be used in a WSDL <portType> definition in a WSDL document.
- All WSDL <operation> elements in a WSDL <portType> element must have names that are unique within that <portType> element.

The <binding> Element

References:
http://www.w3.org/TR/wsdl#_bindings
http://www.w3.org/TR/wsdl#_soap-b
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLBINDINGS
http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDLSOAPBINDING
The <binding> element specifies the following for an abstract interface of a web service (<portType> and <operation> elements):

- The concrete protocols to use, such as HTTP and SOAP.
- The messaging style (RPC or Document).
- The encoding style (Literal or SOAP Encoding).

The <binding> element and its sub elements (marked with purple) are used together with protocol-specific elements (marked with green - SOAP is all we'll see, since only SOAP web services are discussed).

```
    <binding name="BookPrice_Binding" type="sd:BookQuote">
        <!-- RPC over HTTP. -->
        <soapbind:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>

        <!-- Binding for the getBookPrice operation; RPC for both input and output. -->
        <operation name="getBookPrice">
            <soapbind:operation style="rpc"
soapAction="http://www.somedomain.com/xyz/BookQuote/GetBookPrice" />
            <input>
                <soapbind:body use="literal"
                    namespace="http://www.somedomain.com/xyz/BookQuote" />
            </input>
            <output>
                <soapbind:body use="literal" namespace="http://www.somedomain.com/xyz/BookQuote" />
            </output>
        </operation>
    </binding>
```

The <binding> element has two attributes:

- The *name* attribute defines the name of the binding.
  This is later used when a binding is associated with an internet address.
  The name must be unique among the <binding> elements within the enclosing WSDL document.
- The *type* attribute specifies which abstract web service interface, defined in a <portType> element, the binding applies to.

The <soapbind:binding> element is a member of the SOAP-WSDL namespace (http://schemas.xmlsoap.org/wsdl/soap/) and has two attributes:

- The *style* attribute defines the default messaging style for the “methods” in the abstract interface of the web service. Possible values: “rpc” or “document”.
- The *transport* attribute identifies the internet protocol to transfer SOAP messages. For HTTP the value should be “http://schemas.xmlsoap.org/soap/http”.

The <operation> element contains the following definitions for one single method of the web service:

- The name of the method (<operation> *name* attribute).
- Messaging style (<soapbind:operation> *style* attribute).
- Routing information for the operation that can be placed in the SOAPAction HTTP header field (<soapbind:operation> *soapAction* attribute).
- Encoding style and namespace, if RPC is used, of the input message (in the <input> element, the <soapbind:body> *use* and *namespace* attributes).
- Encoding style and namespace, if RPC is used, of the output message (in the <output> element, the <soapbind:body> *use* and *namespace* attributes).

The grammar for a binding is as follows:

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element (1) --> *
        <wsdl:operation name="nmtoken"> *
           <-- extensibility element (2) --> *
           <wsdl:input name="nmtoken"? > ?
               <-- extensibility element (3) -->
           </wsdl:input>
           <wsdl:output name="nmtoken"? > ?
               <-- extensibility element (4) --> *
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <-- extensibility element (5) --> *
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

- A binding must specify exactly one protocol.
- A binding must not specify address information.

The <soapbind:body> Element

References: http://www.w3.org/TR/2001/NOTE-wsdl-20010315#_soap:body

The <soapbind:body> element specifies how the message parts appear in the SOAP <Body> element. The <soapbind:body> element have the following attributes (not exhaustive list):

- The *use* attribute is required to be "literal", since no other encoding types are allowed by the BP. This is also the default value, if the attribute is not declared.
- The *parts* attribute specifies which <part> elements of the input- or output-message of the method is used. This attribute is only needed if a subset of the <part>-s of a message is used.
- The *namespace* attribute is only used with RPC-style messages and must then contain a valid URI, for example the same as the *targetNamespace* of the WSDL document.
  If document-style messaging is used, the *namespace* attribute must not be specified, since the namespace of the XML document fragment being the payload of the message can be derived from its XML schema.

Example of a <soapbind:body> elements used with RPC-style messaging:

```
<operation name="getBookPrice">
    <soapbind:operation style="rpc"
        soapAction="http://www.somedomain.com/xyz/BookQuote/GetBookPrice"/>
        <input>
            <soapbind:body use="literal"
                namespace="http://www.somedomain.com/xyz/BookQuote" />
        </input>
        <output>
            <soapbind:body use="literal"
                namespace="http://www.somedomain.com/xyz/BookQuote" />
        </output>
</operation>
```

Example of a <soapbind:body> elements used with Document-style messaging:

```
<operation name="submit">
    <soapbind:operation style="document"/>
        <input>
            <soapbind:body use="literal" />
        </input>
        <output>
            <soapbind:body use="literal" />
        </output>
</operation>
```

The <soapbind:fault> Element

Reference: http://www.w3.org/TR/2001/NOTE-wsdl-20010315#_soap:fault

In order to specify the encoding for faults, <soapbind:fault> elements are to be declared in the corresponding <operation> element in the <binding> element.

```
…
<binding name="BookPrice_Binding" type="sd:BookQuote">
  <soapbind:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>

  <operation name="getBookPrice">
    <soapbind:operation style="rpc"
     soapAction=
     "http://www.somedomain.com/xyz/BookQuote/GetBookPrice"/>
    <input>
      <soapbind:body use="literal"
       namespace="http://www.somedomain.com/xyz/BookQuote" />
    </input>
    <output>
      <soapbind:body use="literal"
       namespace="http://www.somedomain.com/xyz/BookQuote" />
    </output>
   <fault name="InvalidArgumentFault">
     <soapbind:fault name="InvalidArgumentFault" use="literal" />
    </fault>
  </operation>

</binding>
…
```

The <fault> element and the <soapbind:fault> element are both required to declare a *name* attribute which refers to the name of the fault as defined in the abstract interface (the <portType> element) of the web service.
The *use* attribute of a <soapbind:fault> element must be “literal”. If it is omitted, the default value is “literal”.
Each <operation> element may contain zero or more <fault> elements, each of which contains a <soapbind:fault> element.

The <soapbind:header> Element

Reference: http://www.w3.org/TR/2001/NOTE-wsdl-20010315#_soap:header

Header blocks being received and sent by a web service should also be specified in the <input> and/or the <output> elements of a binding. It is not necessary to specify all headers that may appear, some may be added by intermediaries along the path to the Ultimate Receiver. Specifying header blocks is accomplished by using the <soapbind:header> element.

First, any type(s) that are to be used in the header message must be defined, or imported, in the WSDL <types> element.

```
...
<types>
  <xsd:schema
    targetNamespace="http://www.somedomain.com/xyz/BookQuote"
    xmlns="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="message-id" type="string"/>

  </xsd:schema>
</types>
...
```

Second, a <message> is defined that describes the header's content:

```
<message name="Headers">
  <part name="message-id" element="sd:message-id" />
</message>

...
```

Note that the element attribute in the <part> element must be used, since the “message-id” thing is declared as an <element> in the <types> section above.

Third, the <soapbind:header> element is used in the appropriate <input> and/or <output> element of the <binding> of an <operation>, before the <soapbind:body> element:

```
...
<binding name="BookPrice_Binding" type="sd:BookQuote">
  <soapbind:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getBookPrice">
    <soapbind:operation style="rpc"
     soapAction=
     "http://www.somedomain.com/xyz/BookQuote/GetBookPrice"/>

      <input>
        <soapbind:header message="sd:Headers" part="message-id"
          use="literal"/>
        <soapbind:body use="literal"
          namespace="http://www.somedomain.com/xyz/BookQuote"/>
      </input>

      <output>
        <soapbind:body use="literal"
         namespace="http://www.somedomain.com/xyz/BookQuote"/>
      </output>
  </operation>
</binding>
...
```

Note that, in the <soapbind:header> element:

- The value of the *message* attribute refers to the name of the <message> defined in step two.
- The value of the *part* attribute refers to the <part> of the <message> defined in step two.
- The value of the *use* attribute is always “literal”, whether declared or not.

The <soapbind:headerfault> Element

Reference: http://www.w3.org/TR/2001/NOTE-wsdl-20010315#_soap:header

If there is a response message, any header-related faults must be returned in the <Header> element of the SOAP response message. Such header-related faults must be declared using the <soapbind:headerfault> element inside the <soapbind:header> which may cause the fault:

First, a global type is defined that will hold the header-related fault data:

```
...
<xs:element name="ConfigurationFault" type="tns:ConfigurationFaultType"/>
<xs:complexType name="ConfigurationFaultType">
      <xs:sequence>
            <xs:annotation>
                  <xs:documentation>
                        It is required that every ServiceUrl that is missing,
                        contains a duplicate, or does not resolve to a
                        reachable service will be indicated in the
                        ErroneousElement array.
                  </xs:documentation>
            </xs:annotation>
            <xs:element name="Message" type="xs:string"/>
            <xs:element name="ErroneousElement"
                  type="tns:ConfigurationEndpointRole"
                  minOccurs="0" maxOccurs="8"/>
      </xs:sequence>
      <xs:attribute ref="s:mustUnderstand"/>
</xs:complexType>
...
```

Second, a WSDL message is defined, possibly in another file:

```
...
<wsdl:message name="ConfigurationFaultMessage">
    <wsdl:documentation>
       The Configuration fault header indicates why the configuration was
       invalid and enumerates all of the service URLs which were not valid
       in Configuration header of the request message.
    </wsdl:documentation>
    <wsdl:part
        name="ConfigurationFault"
        element="configurationType:ConfigurationFault"/>
</wsdl:message>
...
```

Finally, using the WSDL <message> name and the <part> name of the message, the <soap:headerfault> can be declared in appropriate <soap:header> element.

```
...
<wsdl:binding name="WarehouseSoapBinding" type="tns:WarehouseShipmentsPortType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
    <wsdl:operation name="ShipGoods">
        <soap:operation
            soapAction="http://www.ws-i.org/SampleApplications/Supply..."/>
        <wsdl:input>
            <soap:body use="literal"
                namespace="http://www.ws-i.org/SampleApplications/Supply..."
                parts="ItemList Customer" />
            <soap:header message="tns:ShipGoodsRequest"
                part="ConfigurationHeader"
                use="literal">
                <soap:headerfault message="c:ConfigurationFaultMessage"
                    part="ConfigurationFault" use="literal" />
            </soap:header>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"
                namespace="http://www.ws-i.org/SampleApplications/Supply..."/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
```

As with <soapbind:header> elements; in the <soapbind:headerfault> element:

- The value of the *message* attribute refers to the name of the <message> defined in step two.
- The value of the *part* attribute refers to the <part> of the <message> defined in step two.
- The value of the *use* attribute is always “literal”, whether declared or not.

The <soapbind:address> Element

Reference: http://www.w3.org/TR/wsdl#_soap:address

The <soapbind:address> element simply assigns an internet address to a <port>, using its *location* attribute. The URI scheme used to specify the address must match the transport type specified in the corresponding <binding> element.
See above section for example usage of the <soapbind:address> element!

Basic Profile on Bindings and SOAP Binding

Definitions:

The BP defines an "operation signature" to be the fully qualified name of the child element of the SOAP <Body> element of the SOAP input message described in a WSDL <operation> element in a WSDL binding.
In the case of RPC/Literal binding, the operation name is used as a wrapper for the part accessors. In the Document/Literal case, since a wrapper with the operation name is not present, the message signatures must be correctly designed so that they meet this requirement.

The BP says:

- A WSDL <binding> element in a WSDL document must use WSDL SOAP binding.
  See http://www.w3.org/TR/2001/NOTE-wsdl-20010315#_soap-b
- The WSDL <binding> element in a WSDL document must contain a <soapbind:binding> element that specifies its *transport* attribute and the value of the *transport* attribute must be "http://schemas.xmlsoap.org/soap/http".
  This requirement does not prohibit the use of HTTPS!
- A WSDL <binding> element in a WSDL document must be either a RPC/Literal binding or a Document/Literal binding.
- A WSDL <binding> element must use literal encoding, that is:
  The *use* attribute of all its <soapbind:body>, <soapbind:fault>, <soapbind:header> and <soapbind:headerfault> child elements must have the value “literal”.
- One or more WSDL documents may contain more than one WSDL <binding> elements that refer to one and the same WSDL <portType> element.
- Each WSDL <operation> element in a WSDL <binding> element in a WSDL document must result in an unique operation signature (see definition above).
- The value of the *location* attribute in a <soapbind:address> element must be unique within a WSDL document. That is, multiple bindings cannot be assigned the same internet address in a WSDL document.
- Messages using Document/Literal binding must be serialized as a SOAP message in which the child element of the SOAP <Body> element is an instance of the global element declaration referenced by the corresponding WSDL <message> element.

- The HTTP response from a web service operation using the one-way message exchange pattern must have an empty body, i.e. it must not contain a SOAP message.
- The client of a web service operation using the one-way message exchange pattern must ignore any SOAP messages returned.
- The client of a web service operation using the one-way message exchange pattern must not interpret a HTTP success response (2xx) as an indication that the message is correct or that the message will be, or has been, processed by the receiver.
  Such a response just indicates that the message has reached the receiver.
- A Document/Literal binding in a WSDL document must not have the *namespace* attribute specified on the following kinds of elements in the binding: <soapbind:body>, <soapbind:fault>, <soapbind:header> and <soapbind:headerfault>.
  Since the namespace can be derived from the XML schema specifying the contents of the above elements.
- An RPC/Literal binding in a WSDL document must have the *namespace* attribute specified on <sopbind:body> elements and the value must be an absolute URI.
  Since the WSDL <part> elements specifying the contents of the message uses the *type* attribute, no namespace can be derived.
- An RPC/Literal binding in a WSDL document must not have the *namespace* attribute specified on the following kids of elements in the binding: <soapbind:fault>, <soapbind:header> and <soapbind:headerfault>.
  Since the namespace can be derived from the XML schema specifying the contents of the above elements.
- A WSDL <binding> element in a WSDL document must have the same set of operations as the WSDL <portType> element to which the binding refers.
- A WSDL <binding> in a WSDL document may contain zero <soapbind:headerfault> elements if there are no known header faults.
- A WSDL <binding> in a WSDL document should contain a <soapbind:fault> element describing all the known faults.
- A WSDL <binding> in a WSDL document should contain a <soapbind:headerfault> element describing all the known header faults.
- A SOAP message may contain a SOAP <Fault> element with a SOAP <Detail> element with a content not described in a <soapbind:fault> element in the corresponding WSDL document.
- A SOAP message may contain information on a fault related to the processing of a header in a SOAP <Header> element that is not described in a <soapbind:headerfault> element in the corresponding WSDL document.
- <soapbind:header> and <soapbind:headerfault> elements in a WSDL <binding> element must use the *part* attribute, not the *parts* attribute.
- All <soapbind:fault> elements in WSDL <binding> elements must have the *name* attribute specified and the value of the attribute must be the same as the *name* attribute of the WSDL <fault> element being the parent of the <soapbind:fault> element.
  The value of the *name* attribute must, in turn, match the value of the *name* attribute of the WSDL <fault> element in the WSDL <portType> to which the binding refers.

- The *use* attribute of the following elements in a WSDL <binding> element in a WSDL document:
  - Has the default value “literal”, if not specified.
  - Must have the value “literal”, if specified.
  The elements are: <soapbind:body>, <soapbind:fault>, <soapbind:header> and <soapbind:headerfault>.
- If a web service server receives a SOAP message that is inconsistent with its WSDL document, it must check for “VersionMismatch”, “MustUnderstand” and “Client” fault conditions in that order.
- If a web service server receives a SOAP message that is inconsistent with its WSDL document, it should generate a SOAP fault with a fault code “Client”, unless a “MustUnderstand” or “VersionMismatch” fault has occurred.
- A SOAP message containing a response to a RPC/Literal request must, in the SOAP <Body> element, contain an element with the name of the invoked operation with the string “Response” appended.
- The elements of a SOAP message with RPC/Literal binding holding the parameters and return value must not be assigned a namespace.
- The elements of a SOAP message with RPC/Literal binding holding the parameters and return value must have the same local names as the *name* attribute of the corresponding WSDL <part> elements.
- The child elements of the elements in a SOAP message with RPC/Literal binding holding the parameters and return value must be prefixed by the appropriate namespace prefix, according to the schema in which they are defined.
  Example: http://www.ws-i.org/Profiles/BasicProfile-1.1.html#Namespaces_for_Children_of_Part_Accessors
- A SOAP message must include header blocks corresponding to all <soapbind:header> elements that are defined in the WSDL <input> and WSDL <output> elements in a WSDL <operation> element in a WSDL <binding> element.
- A SOAP message may contain header block that are not described in the WSDL <binding> element describing the message.
  Such header blocks may have the *mustUnderstand* attribute set to “1”.
- Header blocks in a SOAP message may be arbitrarily ordered, and does not have to consider the ordering of the <soapbind:header> elements in a WSDL document.
- A SOAP message may contain more than one instance of a header block for each <soapbind:header> element in a WSDL document.
- A HTTP request message must contain a SOAPAction header field with a quoted value equal to the value of the *soapAction* attribute of the <soapbind:operation> element, if present in the corresponding WSDL document.
  If the *soapAction* attribute of the <soapbind:operation> element is not specified or equal to the empty string, the SOAPAction header field should be the quoted empty string.
- A SOAP message consumer must understand and process all all WSDL 1.1 SOAP Binding extension elements, regardless of the presence or absence of the WSDL *required* attribute on an extension element and regardless of the value of the *required* attribute, when present.
- A SOAP message consumer must not interpret the presence of the WSDL *required* attribute,

with the value “false”, on an extension element as to mean that the extension element is optional in SOAP messages generated from the WSDL document.
The *required* attribute of extension elements should always be “false” (previously mentioned, see http://www.ws-i.org/Profiles/BasicProfile-1.1.html#WSDL_Extensions).

The <service> and <port> Elements

References:
http://www.w3.org/TR/wsdl#_ports
http://www.w3.org/TR/wsdl#_services

The WSDL <service> element contains one or more <port> elements, each represents a web service. A <port> element assigns an URL to a <binding>. Two, or more, <port> elements may assign different URLs to the same binding, which may be useful for load balancing or failover.

```
...
<service name="BookPriceService">

    <port name="BookPrice_Port" binding="sd:BookPrice_Binding">
        <soapbind:address location="http://www.somedomain.com/xyz/BookQuote" />
    </port>

    <port name="BookPrice_Failover_Port" binding="sd:BookPrice_Binding">
        <soapbind:address location="http://www.somedomain.org/xyz/BookPrice" />
    </port>

    <port name="SubmitPurchaseOrder_Port" binding="sd:SubmitPurchaseOrder_Binding">
        <soapbind:address location="https://www.somedomain.org/xyz/po" />
    </port>

</service>
...
```

The first two <port> elements in the above <service> element assigns two different addresses to the same binding (BookPrice_Binding) using the <soapbind:address> element.
The third <port> element assigns an address to the SubmitPurchaseOrder_Binding binding. Note that the HTTPS protocol is to be used.

The <port> element has two attributes:

- name
  Defines a name of the port. The name must be unique among <port> elements in the enclosing WSDL document.
- binding
  A reference to the <binding> for which the <port> assigns an address.

A <port> must specify exactly one address for a <binding> and may not specify any additional binding information.

The <service> element has one single attribute:

- name
  Defines the name of the service. The name must be unique among <service> elements in the enclosing WSDL document.

WSDL Binding Mechanisms

WSDL defines a common binding mechanism. This is used to attach a specific protocol or data format or structure to an abstract message, operation, or endpoint. It allows the reuse of abstract definitions.

WSDL includes a binding for SOAP 1.1 endpoints, which supports the specification of the following protocol specific information:

- An indication that a binding is bound to the SOAP 1.1 protocol.
- A way of specifying an address for a SOAP endpoint.
- The URI for the SOAPAction HTTP header for the HTTP binding of SOAP
- A list of definitions for Headers that are transmitted as part of the SOAP Envelope

For more information on the WSDL binding for SOAP, see the section on the WSDL <binding> element and the section on the WSDL <service> and <ports> elements above.

Basic WSDL Operation Types

WSDL has four transmission primitives that an endpoint can support:

- One-way. The endpoint receives a message.
- Request-response. The endpoint receives a message, and sends a correlated message.
- Solicit-response. The endpoint sends a message, and receives a correlated message.
- Notification. The endpoint sends a message.

WSDL refers to these primitives as operations. Although the base WSDL structure supports bindings for these four transmission primitives, WSDL only defines bindings for the One-way and Request-response primitives.

See the above section on the WSDL <portType> and <operation> elements for details on the different operation types.

3.2 WSDL Abstract vs Concrete

Describe how WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as "how" and "where" that functionality is offered.

Reference: http://www.w3.org/TR/wsdl#_style

Using the WSDL <import> statement, different elements of a WSDL document may be separated into independent documents. This helps writing more clear web service definitions and enables possible reuse of different parts of a service definition.
As an example from the WSDL specification document, a WSDL document can be split in three different parts:

- Data type definitions.
  One or more XML schemas defining custom types are created.
- Message definitions and the abstract interface(s) of the web service.
  XML schema(s) defining custom data types needed are imported in the WSDL <types> element.
  One or more WSDL <message> elements and WSDL <portType> elements are defined in this WSDL document.
- Specific service bindings.
  WSDL document(s) defining message(s) and abstract interface(s) are imported using the WSDL <import> element.
  Abstract interface(s) (WSDL <portType> element(s)) are then bound in the WSDL <binding> element(s) and each binding is then assigned an endpoint address in WSDL <service> element(s).

The data type, message and abstract interface definitions are the abstract definitions. The bindings are the concrete part.

3.3 WSDL Component Model

Describe the Component Model of WSDL including Descriptions, Interfaces, Bindings, Services and Endpoints.

Reference: http://www.w3.org/TR/2007/REC-wsdl20-20070626/#component_model

Note that while the above sections described WSDL 1.1, the WSDL component model was introduced in WSDL 2.0, which is the version that this section concerns itself with.
Note also that a certain amount of details has deliberately been left out of this section. Please consult the references for more complete information!

The WSDL 2.0 component model introduces components as a way of describing different aspects of a web service. A component is a typed collection of properties which may or may not have an XML representation. The component model allows WSDL 2.0 definitions to be independent of any particular serialization, including XML.

- Description
- Interface
- Interface Fault
- Interface Operation
- Interface Message Reference
- Interface Fault Reference
- Binding
- Binding Fault
- Binding Operation
- Binding Message Reference
- Binding Fault Reference
- Service
- Endpoint

The sections below describes the different types of components, their different properties and how each component is represented using XML 1.0.

The Description Component

Reference: http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Description

The Description Component is a container for the following types of components:

Component	Comments	XML Representation
Interface	WSDL 2.0 component.	<interface>
Binding	WSDL 2.0 component.	<binding>
Service	WSDL 2.0 component.	<service>
Element Declaration	Type system component.	<import> and <include>
Type Definitions	Type system component.	<types>

The XML representation of the Description Component looks like this:

```
<description targetNamespace="xs:anyURI" >
  <documentation />*
  [ <import /> | <include /> ]*
  <types />?
  [ <interface /> | <binding /> | <service /> ]*
</description>
```

All components defined in the same description component belongs to the same target namespace. Component names must be unique within the component type.

The <description> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- One required *targetNamespace* attribute.
  Defines the namespace to which all components in the WSDL document belongs.
- Zero or more XML namespace definitions whose value is not “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements, in any order, from the following elements:
  - Zero or more <include> elements.
  - Zero or more <import> elements.
  - Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.
- One optional <types> element.
- Zero or more elements, in any order, from the following elements:
  - <interface> element.
  - <binding> element.
  - <service> element.
  - Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what a Description Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/ns/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/ns/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsdlx= "http://www.w3.org/ns/wsdl-extensions">

    …
</description>
```

Element Declarations

Element Declaration Components are the XML elements available in a WSDL definition from the following sources:

- Element definitions in the Type Definitions Component (the WSDL <types> element).
- Element definitions from other XML schemas included or imported, using the XML <include> and/or XML <import> elements.
  This will, at a minimum, consist of all global element declarations in the imported/included XML schemas.

Each XML schema element declaration must have an unique qualified name.

Type Definitions

Type Definition Components are XML type definitions from the following sources:

- Type definitions in the Type Definitions Component (the WSDL <types> element).
- Type definitions from other XML schemas included or imported, using the XML <include> and/or XML <import> elements.
- The following built-in data types defined by XML schema:
  xs:string, xs:boolean, xs:decimal, xs:float, xs:double, xs:duration, xs:dateTime, xs:time, xs:date, xs:gYearMonth, xs:gYear, xs:gMonthDay, xs:gDay, xs:gMonth, xs:hexBinary, xs:base64Binary, xs:anyURI, xs:QName, xs:NOTATION, and the twenty-five derived datatypes xs:normalizedString, xs:token, xs:language, xs:NMTOKEN, xs:NMTOKENS, xs:Name, xs:NCName, xs:ID, xs:IDREF, xs:IDREFS, xs:ENTITY, xs:ENTITIES, xs:integer, xs:nonPositiveInteger, xs:negativeInteger, xs:long, xs:int, xs:short, xs:byte, xs:nonNegativeInteger, xs:unsignedLong, xs:unsignedInt, xs:unsignedShort, xs:unsignedByte, xs:positiveInteger.

Each XML schema type declaration must have an unique qualified name.

Here is an example of what a Description Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>

  <documentation>
    Blah blah
  </documentation>

  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

      <xs:element name="checkAvailability" type="tCheckAvailability"/>
      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="checkAvailabilityResponse" type="xs:double"/>

      <xs:element name="invalidDataError" type="xs:string"/>

    </xs:schema>
  </types>
  ...
</description>
```

The Interface Component

Reference: http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Interface

An Interface Component consists of a set of operations. Each operation defines the optional input message that is sent when invoking the operation and the optional output message produced by the operation.
New in WSDL 2.0 is that an interface can extend one or more other interfaces. Circular references are not allowed, so the current interface must not, either directly or indirectly, appear in the set of interfaces it extends. Identical Operation Components (operations) that appear in more than one interface are treated as a single Operation Component.
The Interface Component is a container for the following types of components:

Component	Comments	XML Representation
Interface Fault	Describes fault(s) that may occur during invocation of operations in the interface.	<fault>
Interface Operation	Defines operation(s) available in the interface.	<operation>

The XML representation of the Interface Component looks like this:

```
<description>
  <interface
        name="xs:NCName"
        extends="list of xs:QName"?
        styleDefault="list of xs:anyURI"? >
    <documentation />*
    [ <fault /> | <operation /> ]*
  </interface>
</description>
```

The <interface> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements, in any order, from the following elements:
  - Zero or more <fault> elements.
  - Zero or more <operation> elements.
  - Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

The <interface> element can have the following attributes:

Attribute Name	Required	Description
name	Yes	Name of the Interface Component.
extends	No	Space-separated list of the namespace qualified names of the interfaces this Interface Component extends. The list must not contain duplicates.
styleDefault	No	An URI defining the default operation style for the Operation Components of this Interface Component. See below!

Here is an example of what an Interface Component, including an Interface Fault and an Interface Operation subcomponent, may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <interface name="reservationInterface" >

    <fault name="invalidDataFault" element="ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
        <input messageLabel="In" element="ghns:checkAvailability" />
        <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>
    ...
  </interface>
  …
</description>
```

The Interface Fault Component

Reference: http://www.w3.org/TR/2007/REC-wsdl20-20070626/#InterfaceFault

Interface Fault Components are something like abstract exception definitions. The name and the content of the fault is defined, but the fault is not associated with any operations yet.
This enables us to quickly overview the faults that can be generated by all the operations in an interface of a web service. We can also define multiple operations that possibly generate the same kind of fault without having to re-define the contents of the fault in each operation.
The faults defined by the Interface Fault Component(s) in an Interface Component are application specific faults that clients of the web service are to expect and possibly be able to recover from. Other kinds of faults may also be generated at runtime when invoking operations in the interface, such as faults caused by network problems, out of memory conditions or other system faults.

The XML representation of the Interface Fault Component looks like this:

```
<description>
  <interface>
    <fault name="xs:NCName" element="union of xs:QName, xs:token"? >
      <documentation />*
    </fault>
  </interface>
</description>
```

The <fault> element can have the following attributes:

Attribute Name	Required	Description
name	Yes	Name of the fault. Must be unique within the enclosing Interface Component.
element	No	A reference to a global XML element declaration describing the contents of the fault.

The <fault> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what an Interface Fault Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <interface name="reservationInterface" >

    <fault name="invalidDataFault" element="ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
        ...
    </operation>
    ...
  </interface>
  …
</description>
```

The Interface Operation Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#InterfaceOperation
http://www.w3.org/TR/wsdl20-adjuncts/#meps
http://www.w3.org/TR/wsdl20-adjuncts/#styles
http://www.w3.org/TR/wsdl20-adjuncts/#safety
http://www.w3.org/TR/2004/REC-webarch-20041215/#safe-interaction

An Interface Operation Component is similar to a method definitions in a Java interface. It defines the data that is to be supplied to the operation, the data that will be produced by the operation and any faults that may occur during the execution of the operation.
An Interface Operation Component is also able to specify additional properties of an operation, such as message exchange pattern and operation style (RPC or Document).

The XML representation of the Interface Operation Component without any extensions looks like this:

```
<description>
  <interface>
    <operation name="xs:NCName" pattern="xs:anyURI"? style="list of xs:anyURI"? >
      <documentation />*
      [ <input /> | <output /> | <infault /> | <outfault /> ]*
    </operation>
  </interface>
</description>
```

The <operation> element can have the following attributes:

Attribute Name	Required	Description
name	Yes	Name of the operation. Must be unique within the enclosing Interface Component.
pattern	No	URI defining the message exchange pattern of the operation.
style	No	URI defining the operation style.
wsdlx:safe	No	If true, then the operations is idempotent, if false then the operation may or may not be idempotent. Default value is false.

The following predefined values exists for the *pattern* attribute:

Message Exchange Pattern Name	URI
In-Out	http://www.w3.org/ns/wsdl/in-out
In-Only	http://www.w3.org/ns/wsdl/in-only
Robust In-Only	http://www.w3.org/ns/wsdl/robust-in-only

The following predefined values exists for the style attribute:

Operation Style	URI
RPC	http://www.w3.org/ns/wsdl/style/rpc
IRI	http://www.w3.org/ns/wsdl/style/iri
Multipart	http://www.w3.org/ns/wsdl/style/multipart

The <operation> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- One or more elements, in any order, from the following elements:
  - Zero or more <input> elements.
  - Zero or more <output> elements.
  - Zero or more <infault> elements.
  - Zero or more <outfault> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what an Interface Operation Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <interface name="reservationInterface" >

    <fault name="invalidDataFault" element="ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">

        <input messageLabel="In" element="ghns:checkAvailability" />
        <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>

    </operation>
    ...
  </interface>
  …
</description>
```

Interface Message Reference Component

Reference: http://www.w3.org/TR/2007/REC-wsdl20-20070626/#InterfaceMessageReference

Interface Message Reference Components describes the contents of a message sent to, or received from, an operation in a web service by referring to a definition usually made using XML.
Compare with the parameters of a Java method, which can be seen as the input message of that method, and the return value, which can be seen as the output message.
Additionally, web service operations can also have out-parameters, which are parameters that acts as a receiver of some result produced by an operation, and in-out parameters, which allows for a parameter to be passed to the operation as well as a result to be stored in the parameter and later retrieved by the client of the operation.

The XML representation of the Interface Message Reference Component without any extensions looks like this:

```
<description>
  <interface>
    <operation>
      <input
            messageLabel="xs:NCName"?
            element="union of xs:QName, xs:token"? >
        <documentation />*
      </input>
      <output
            messageLabel="xs:NCName"?
            element="union of xs:QName, xs:token"? >
        <documentation />*
      </output>
    </operation>
  </interface>
</description>
```

The <input> and <output> elements can have the following attributes:

Attribute Name	Required	Description
messageLabel	No	Describes the role the message plays in the message exchange pattern of the operation. Value must match those defined for the message exchange pattern used. Each kind of value of this attribute must at most occur once inside an Interface Operation Component. Possible values for predefined message exchange patterns are: In, Out
element	No	A reference to an Element Declaration Component describing the contents of the message. Must refer to a global element declaration in an XML schema.

The *messageLabel* attribute is not needed for the predefined WSDL message exchange patterns when the operation has only one message with a given direction.

An <input> or <output> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Not all message exchange patterns allow for all types of Interface Message Reference Components; for instance the In-Only message exchange pattern does not allow for use of an Interface Message Reference Component represented by an <output> element.

For an example of what Interface Message Reference Components looks like in a WSDL document, please refer to the above section showing an example of the Interface Component.
Here is an example of what an Interface Message Components may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <interface name="reservationInterface" >

    <fault name="invalidDataFault" element="ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">

        <input messageLabel="In" element="ghns:checkAvailability" />
        <output messageLabel="Out" element="ghns:checkAvailabilityResponse" />

        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>
    ...
  </interface>
  …
</description>
```

The Interface Fault Reference Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#InterfaceFaultReference
http://www.w3.org/TR/wsdl20-adjuncts/#fault-rules

An Interface Fault Reference Component describes a fault that may occur during a web service operation. The description of the fault is a reference to an Interface Fault Component defined for the Interface Component in which the operation occur.
Fault propagation rules describes how a fault that has occurred in connection to a message should be transmitted. There are three different fault propagation rules defined by the WSDL 2.0 standard:

Rule Name	Description	URI
Fault Replaces Message	Any message after the first message in the message exchange pattern in question may be replaced by a fault. The fault must have the same direction and must be delivered to the same target as as the message it replaces. If the fault cannot be delivered, it must be discarded.	http://www.w3.org/ns/wsdl/fault-replaces-message
Message Triggers Fault	Any message in the message exchange pattern may trigger a fault. The fault must have the opposite direction of the message that triggered it and it must be delivered to the node from which the message triggering the fault originated. Any node may pass the fault message on to another node, but must only do this once for each triggering message.	http://www.w3.org/ns/wsdl/message-triggers-fault
No Faults	No faults are allowed to be sent.	http://www.w3.org/ns/wsdl/no-faults

The XML representation of the Interface Fault Reference Component looks like this:

```
<description>
  <interface>
    <operation>
      <infault
            ref="xs:QName"
            messageLabel="xs:NCName"? >
        <documentation />*
      </infault>*
      <outfault
            ref="xs:QName"
            messageLabel="xs:NCName"? >
        <documentation />*
      </outfault>*
    </operation>
  </interface>
</description>
```

The <infault> and <outfault> elements can have the following attributes:

Attribute Name	Required	Description
ref	Yes	Namespace qualified reference to an Interface Fault Component.
messageLabel	Yes, if the message exchange pattern used by the Interface Operation Component in which the Interface Fault Reference Component occurs has more than one message with the direction of the Otherwise not required.	Describes the role the message delivering the fault plays in the message exchange pattern of the operation. Value must match those defined for the message exchange pattern used. Possible values for predefined message exchange patterns are: In, Out

An <infault> or <outfault> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what an Interface Fault Reference Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <interface name="reservationInterface" >

    <fault name="invalidDataFault" element="ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
        <input .../>
        <output .../>

        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>

    </operation>
    ...
  </interface>
  …
</description>
```

The Binding Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Binding
http://www.w3.org/TR/wsdl20-adjuncts/#soap-binding
http://www.w3.org/TR/wsdl20-adjuncts/#http-binding

In this section, the Binding Component of a WSDL document used with SOAP and HTTP bindings will be described. No other kind of bindings will be considered. The SOAP binding extension is SOAP version independent.
A Binding Component defines the implementation details necessary to access a web service. Such details include the concrete format of messages and which transmission protocol is to be used.
A Binding Component can specified in two different ways:

- To Enable Reuse
  - The Binding Component does not specify any operation-specific details.
  - The Binding Component does not specify any fault binding details.
  - The Binding Component does not specify an interface to which it applies.
- To Apply to One Single, Specific, Interface
  - The Binding Component specifies operation-specific details.
  or
  - The Binding Component specifies fault binding details.
  If none of the above are true then the Binding Component may optionally specify an interface to which it applies.
  If any of the above is true, then the Binding Component must specify which interface to which it applies.

If the Binding Component is specified in a way that enables reuse, then the Binding Component can be used in connection with one or more interfaces.
If the Binding Component has specified an interface to which it applies, then it cannot be applied to other interfaces and thus cannot be reused.

Non-Reusable Bindings
A Binding Component that specifies to which Interface Component it applies must define bindings for all operations (this includes any faults of the operations) in the Interface Component; either by defaulting rules or by defining bindings for each Interface Operation and Interface Fault component in the interface.
The Binding Component must also define bindings for all faults, that are referenced from operation(s), in the Interface Component.

Reusable Bindings
When a Binding Component that does not specify an Interface Component is associated with an Interface Component, which is done in an Endpoint Component, the binding must define bindings for all Interface Operation components and Interface Fault components of the Interface Component. This is accomplished using properties defined in the Binding Component and default binding rules specific to the binding type.

The XML representation, including the SOAP and HTTP binding extensions, of the Binding Component looks like this:

```
<description>
  ...
  <binding name="xs:NCName" interface="xs:QName"?
           type="http://www.w3.org/ns/wsdl/soap"
           whttp:methodDefault="xs:string"?
           whttp:queryParameterSeparatorDefault="xs:string"??
           whttp:contentEncodingDefault="xs:string"??
           whttp:cookies="xs:boolean"?
           wsoap:version="xs:string"?
           wsoap:protocol="xs:anyURI"
           wsoap:mepDefault="xs:anyURI"? >
    <documentation />*
    <wsoap:module ref="xs:anyURI" required="xs:boolean"? >
      <documentation />*
    </wsoap:module>*

    <fault ref="xs:QName"
           wsoap:code="union of xs:QName, xs:token"?
           wsoap:subcodes="union of (list of xs:QName), xs:token"?
           whttp:code="union of xs:int, xs:token"?
           whttp:contentEncoding="xs:string"?? >
      <documentation />*
      <wsoap:module ... />*
      <wsoap:header element="xs:QName" mustUnderstand="xs:boolean"?
                    required="xs:boolean"? >
        <documentation />*
      </wsoap:header>*
      <whttp:header name="xs:string" type="xs:QName"
                    required="xs:boolean"? >
        <documentation />*
      </whttp:header>*
    </fault>*
    <operation ref="xs:QName"
               whttp:location="xs:anyURI"?
               whttp:method="xs:string"?
               whttp:inputSerialization="xs:string"?
               whttp:outputSerialization="xs:string"?
               whttp:faultSerialization="xs:string"?
               whttp:queryParameterSeparator="xs:string"?
               whttp:contentEncodingDefault="xs:string"?
               whttp:ignoreUncited="xs:boolean"?
               wsoap:mep="xs:anyURI"?
               wsoap:action="xs:anyURI"? >
      <documentation />*
      <wsoap:module ... />*
      <input messageLabel="xs:NCName"?
             whttp:contentEncoding="xs:string"? >
        <documentation />*
        <wsoap:module ... />*
        <wsoap:header ... />*
        <whttp:header ... />*
      </input>*
      <output messageLabel="xs:NCName"?
              whttp:contentEncoding="xs:string"? >
        <documentation />*
        <wsoap:module ... />*
        <wsoap:header ... />*
        <whttp:header ... />*
      </output>*
      <infault ref="xs:QName"
                  messageLabel="xs:NCName"?>
        <documentation />*
        <wsoap:module ... />*
      </infault>*
      <outfault ref="xs:QName"
                   messageLabel="xs:NCName"?>
        <documentation />*
        <wsoap:module ... />*
      </outfault>*
    </operation>*
  </binding>
  ...
</description>
```

The namespaces of the SOAP and HTTP extension attributes are “http://www.w3.org/ns/wsdl/soap” and “http://www.w3.org/ns/wsdl/http” respectively.

The <binding> element can have the following attributes, including those added by the SOAP and HTTP extensions:

Attribute Name	Required	Description
name	Yes	With the target namespace of the WSDL document, the value of the *name* attribute forms the qualified name which can be used to refer to the binding.
interface	No	The namespace qualified name of the Interface Component to which the binding applies.
type	Yes	An URI defining the type of binding. For SOAP, the URI is “http://www.w3.org/ns/wsdl/soap”.
whttp:methodDefault		Default HTTP request method for all the Binding Operation Components in the Binding Component.
whttp:queryParameterSeparatorDefault	No	Default HTTP query parameter separator character for all Binding Operation Components in the Binding Component. Default is '&'.
whttp:contentEncodingDefault	No	The value that the HTTP Content-Encoding header will have for all Binding Fault Components and Binding Operation Components unless they specify otherwise.
whttp:cookies	No	If true, then the server relies on HTTP cookies and the client is required to understand them. Default: False.
wsoap:version	No	SOAP version string. Default: “1.2”
wsoap:protocol	Yes	URI specifying the underlying protocol. For HTTP the URI is “http://www.w3.org/2003/05/soap/bindings/HTTP/”.
wsoap:mepDefault	No	Specifies the default message exchange pattern for all Interface Operation Components in the Interface Component to which the binding applies.

A < binding> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements, in any order, from among the following elements:
  - Zero or more <fault> elements.
  - Zero or more <operation> elements.
  - Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what a Binding Component may look like in an actual WSDL document:

```
<description ...>
  ...

  <binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
          type="http://www.w3.org/ns/wsdl/soap"
          wsoap:version="1.2"
          wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap:Sender"/>

    <operation ref="tns:opCheckAvailability"
              wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response">
        <input />
        <output />
        <outfault ref="tns:invalidDataFault" />
    </operation>

  </binding>

  ...
</description>
```

Here is another example of a Binding Component in a WSDL 2.0 document used to describe a RESTful web service:

```
...
<wsdl:binding name="Artist" interface="Artist" http:defaultMethod="GET">
    <!-- GET /Music/Artist?genre="electronica" -->

    <!-- GET /Music/Artist -->
    <wsdl:operation ref="ArtistSearch" location="Artist/{Name}" />

    <!-- GET /Music/ArtistIDRef?genre="electronica"-->
    <wsdl:operation ref="ArtistNameIDSearch" location="ArtistNameID/{Name}" />

    <!-- GET /Music/ArtistNameRef?genre="electronica" -->
    <wsdl:operation ref="ArtistNameRefSearch" location="ArtistNameRef/{Name}" />

    <!-- POST /Music/Artist -->
    <wsdl:operation ref="AddArtistServerID" location="Artist" method="POST" />

    <!-- POST /Music/Artist/7 -->
    <wsdl:operation ref="AddArtistClientID" location="Artist" method="POST" />

    <!-- DELETE /Music/Artist/5 -->
    <wsdl:operation ref="DeleteArtist" location="Artist/{ID}" method="DELETE" />

    <!-- PUT /Music/Artist/5 -->
    <wsdl:operation ref="UpdateArtistWithID" location="Artist/{ID}" method="PUT" />

</wsdl:binding>
...
```

The SOAP Header Block Component

References:
http://www.w3.org/TR/wsdl20-adjuncts/#soap-header-decl-property

A SOAP Header Block Component describes a SOAP header block (see here and here) that is associated to one of the following:

- An input message of an operation.
- An output message from an operation.
- A fault occurring as the result of invoking an operation.

The SOAP Header Block Component is part of the SOAP binding extension to WSDL.
The XML representation of the SOAP Header Block Component looks like this. Note the different locations at which SOAP Header Block Components may occur.

```
<description>
  ...
  <binding name="xs:NCName" type="http://www.w3.org/ns/wsdl/soap" >
    <fault ref="xs:QName" >
      <wsoap:header element="xs:QName" mustUnderstand="xs:boolean"?
                    required="xs:boolean"? >
        <documentation />*
      </wsoap:header>*
      ...
    </fault>*
    <operation ref="xs:QName" >
      <input messageLabel="xs:NCName"?>
        <wsoap:header ... />*
        ...
      </input>*
      <output messageLabel="xs:NCName"?>
        <wsoap:header ... />*
        ...
      </output>*
    </operation>*
  </binding>
  ...
</description>
```

The <wsoap:header> element belongs to the “http://www.w3.org/ns/wsdl/soap” namespace and can have the following attributes:

Attribute Name	Required	Description
element	Yes	Reference to global XML element describing the contents of the header block.
mustUnderstand	No	Default: False.
required	No	Default: False.

A < wsoap:header> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl/soap”.
- Zero or more <documentation> elements.
- Zero or more elements which does not belong to the namespaces “http://www.w3.org/ns/wsdl” or “http://www.w3.org/ns/wsdl/soap”.

Here is an example of what a SOAP Header Block Component may look like in an actual WSDL document:

```
    <binding name="binding1"
      interface="tns:interface1"
      type="http://www.w3.org/ns/wsdl/soap">

        <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>

        <operation
          ref="tns:opCheckAvailability"
          wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response">
            <input>
                <wsoap:header element="ghns:checkAvailability" mustUnderstand="true" />
                <wsoap:header element="ghns:checkPrice" mustUnderstand="true" />
            </input>

            <output>
                <wsoap:header element="ghns:checkAvailability" mustUnderstand="false" />
            </output>
            <outfault ref="tns:invalidDataFault" />
       </operation>
    </binding>
```

The HTTP Header Component

References:
http://www.w3.org/TR/wsdl20-adjuncts/#http-header-decl-property

A HTTP Header Component enables the specification of name and type of value of a HTTP header associated with a web service message. The HTTP Header Component is part of the HTTP binding extension to WSDL.

The XML representation of the HTTP Header Component looks like this. Note the different locations at which HTTP Header Components may occur.

```
<description>
  <binding name="xs:NCName" type="http://www.w3.org/ns/wsdl/http" >
    <fault ref="xs:QName">
      <whttp:header name="xs:string" type="xs:QName" required="xs:boolean"? >
        <documentation />*
      </whttp:header>*
      ...
    </fault>*
    <operation ref="xs:QName" >
      <input messageLabel="xs:NCName"?>
        <whttp:header ... />*
        ...
      </input>*
      <output messageLabel="xs:NCName"?>
        <whttp:header ... />*
        ...
      </output>*
    </operation>*
  </binding>
</description>
```

The <whttp:header> element belongs to the “http://www.w3.org/ns/wsdl/http” namespace and can have the following attributes:

Attribute Name	Required	Description
name	Yes	Name of HTTP header.
type	Yes	Type of value the HTTP header is to contain.
required	No	Indicates whether the HTTP header is required or not. Default: false

A < whttp:header> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl/http”.
- Zero or more <documentation> elements.
- Zero or more elements which does not belong to the namespaces “http://www.w3.org/ns/wsdl” or “http://www.w3.org/ns/wsdl/http”.

Here is an example of what a HTTP Header Component may look like in an actual WSDL document:

```
...
<binding
  name="EchoHTTPBinding"
  interface="tns:Echo"
  type="http://www.w3.org/ns/wsdl/http"
  whttp:cookies="true"
  whttp:methodDefault="PUT">

  <fault ref="tns:EchoNameFault" whttp:code="402">
    <whttp:header name="X-WSDLTestHeader" type="xs:string" required="true"/>
  </fault>

  <operation ref="tns:EchoName" whttp:method="POST" whttp:location="EchoName">
    <input>
      <whttp:header name="X-WSDLTestHeader" type="xs:string" required="true"/>
    </input>
    <output>
      <whttp:header name="X-WSDLTestHeader" type="xs:string" required="true"/>
    </output>
    <outfault ref="tns:EchoNameFault" />
  </operation>
</binding>
...
```

The Binding Fault Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Binding_Fault
http://www.w3.org/TR/wsdl20-adjuncts/#soap-fault-decl
http://www.w3.org/TR/wsdl20-adjuncts/#http-fault-decl

A Binding Fault Component specifies the binding for an Interface Fault Component, that is, how the fault will be formatted and transported when it occurs as part of a message exchange.
It also allows for additional information related to the fault to be specified, such as a SOAP fault code and fault subcode, when SOAP is used.

The XML representation, including the SOAP and HTTP binding extensions, of the Binding Fault Component looks like this:

```
<description>
  ...
  <binding ...>

    <fault ref="xs:QName"
           wsoap:code="union of xs:QName, xs:token"?
           wsoap:subcodes="union of (list of xs:QName), xs:token"?
           whttp:code="union of xs:int, xs:token"?
           whttp:contentEncoding="xs:string"? >
      <documentation />*
      <wsoap:module ... />*
      <wsoap:header element="xs:QName" mustUnderstand="xs:boolean"?
                    required="xs:boolean"? >
        <documentation />*
      </wsoap:header>*
      <whttp:header name="xs:string" type="xs:QName"
                    required="xs:boolean"? >
        <documentation />*
      </whttp:header>*
    </fault>*

    ...
  </binding>

  ...
</description>
```

The <fault> element can have the following attributes, including attributes added by the SOAP and HTTP extensions:

Attribute Name	Required	Description
ref	Yes	Reference to the Interface Fault Component to which the binding applies.
wsoap:code	No	SOAP fault code, see above!
wsoap:subcode	No	SOAP fault subcode, see above!
whttp:code	No	HTTP error code that will be returned by web service when the fault the binding applies to occurs.
whttp:contentEncoding	No	Value the HTTP Content-Encoding header will have when the fault the binding applies to occur.

The namespaces of the SOAP and HTTP extension attributes are “http://www.w3.org/ns/wsdl/soap” and “http://www.w3.org/ns/wsdl/http” respectively.

A < fault> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more <wsoap:header> elements.
- Zero or more <whttp:header> elements.
- Zero or more elements that does not belong to the namespace “http://www.w3.org/ns/wsdl”. Such elements are considered to be binding fault extension elements.

Here is an example of what a Binding Fault Component may look like in an actual WSDL document:

```
<description ...>
  ...
  <binding ...>
    <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>
    ...
  </binding>
  ...
</description>
```

The Binding Operation Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Binding_Operation
http://www.w3.org/TR/wsdl20-adjuncts/#soap-operation-decl
http://www.w3.org/TR/wsdl20-adjuncts/#http-operation-decl

A Binding Operation Component specifies the following properties for a specific Interface Operation Component:

- Partial URI used to access a resource.
- HTTP method used to access a resource.
- Data serialization formats.
- HTTP query parameter separator character.
- A message exchange pattern to be used by the operation.
- Routing information (related to the SOAP Action HTTP header).

WSDL 2.0 can be used not only to describe SOAP web services, but also REST web services, thus the plethora of properties of the Binding Operation Component.
The XML representation, including the SOAP and HTTP binding extension, of the Binding Operation Component looks like this:

```
<description>
  ...
  <binding wsoap:mepDefault="xs:anyURI"? >
    <operation ref="xs:QName"
               whttp:location="xs:anyURI"?
               whttp:method="xs:string"?
               whttp:inputSerialization="xs:string"?
               whttp:outputSerialization="xs:string"?
               whttp:faultSerialization="xs:string"?
               whttp:queryParameterSeparator="xs:string"?
               wsoap:mep="xs:anyURI"?
               wsoap:action="xs:anyURI"? >
      <documentation />*
      [
        <input>
          <wsoap:header ... />*
          <whttp:header ... />*
        </input>
        |
        <output>
          <wsoap:header ... />*
          <whttp:header ... />*
        </output>
        |
        <infault />
        |
        <outfault />
      ]*
    </operation>
    ...
  </binding>
  ...
</description>
```

The <operation> element can have the following attributes, including attributes added by the SOAP extension:

Attribute Name	Required	Description
ref	Yes	Reference to the Interface Operation Component to which the binding applies.
whttp:location	No	Defines a partial URI of the resource (operation) in question. Used when defining RESTful web services.
whttp:method	No	Defines the HTTP method used when accessing the resource (operation). Used when defining RESTful web services.
whttp:inputSerialization	No	Serialization format of data received by the operation. See http://www.w3.org/TR/wsdl20-adjuncts/#_http_binding_default_rule_psf.
whttp:outputSerialization	No	Serialization format of data produced by the operation. See http://www.w3.org/TR/wsdl20-adjuncts/#_http_binding_default_rule_psf.
whttp:faultSerialization	No	Serialization format of faults produced by the operation. See http://www.w3.org/TR/wsdl20-adjuncts/#_http_binding_default_rule_psf. Default: application/xml
whttp:queryParameterSeparator	No	HTTP query parameter separator character. Default: “&”
wsoap:mep	No	Specifies the message exchange pattern for the Interface Operation Component to which the binding applies.
wsoap:action	No	Routing information for the operation that can be placed in the SOAPAction HTTP header field

The namespace of the SOAP and HTTP extension attributes are “http://www.w3.org/ns/wsdl/soap” and “http://www.w3.org/ns/wsdl/http” respectively.

An <operation> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements from the following, in any order:
  - Zero or more <input> elements.
  - Zero or more <output> elements.
  - Zero or more <infault> elements.
  - Zero or more <outfault> elements.
- Zero or more elements that does not belong to the namespace “http://www.w3.org/ns/wsdl”.

The Binding Message Reference Component

Reference:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Binding_Message_Reference

The Binding Message Reference Component assigns a concrete message format to a message in an operation. A Binding Message Reference Component may contain one or more SOAP Header Block Components and/or HTTP Header Components, which are not shown in this section.
The XML representation of the Binding Message Reference Component looks like this:

```
<description>
  ...
  <binding>
    ...
    <operation>
      <input messageLabel="xs:NCName"? >
        <documentation />*
      </input>
      <output messageLabel="xs:NCName"? >
        <documentation />*
      </output>
    </operation>
    ...
  </binding>
  ...
</description>
```

The <input> and <output> elements have a single optional attribute named *messageLabel*. This attribute identifies the role of this message in the message exchange pattern of the Interface Operation Component. Possible values for predefined message exchange patterns are: In, Out.
The *messageLabel* attribute:

- Is required if the operation has more than one placeholder message with the same direction.
- If present, the value must match that of an Interface Message Reference Component.
- If absent, there must be an unique placeholder message with the same direction as the element in which the *messageLabel* attribute appears.

The *messageLabel* attribute is not needed for the predefined WSDL message exchange patterns when the operation has only one message with a given direction.

An <input> or <output> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements that does not belong to the namespace “http://www.w3.org/ns/wsdl”.

The Binding Fault Reference Component

Reference:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Binding_Fault_Reference

The Binding Fault Reference Component assigns a concrete message format to a fault in an operation.
The XML representation of the Binding Fault Reference Component looks like this:

```
<description>
  <binding>
    <operation>
      <infault ref="xs:QName" messageLabel="xs:NCName"?>
        <documentation />*
      </infault>
      <outfault ref="xs:QName" messageLabel="xs:NCName"?>
        <documentation />*
      </outfault>
    </operation>
  </binding>
</description>
```

The <infault> and <outfault> elements can have the following attributes:

Attribute Name	Required	Description
ref	Yes	Reference to the Interface Fault Reference Component to which the binding applies.
messageLabel	No	Identifies the role of this message in the message exchange pattern of the Interface Operation Component. Possible values for predefined message exchange patterns are: In, Out.

Each value of the *ref* attribute within one Binding Operation Component must be unique. This means that the same fault cannot be bound twice within an operation.

The *messageLabel* attribute:

- Is required if the operation has more than one placeholder message with the same direction.
- If present, the value must match that of some placeholder message with the same direction as the element in which the *messageLabel* attribute appears.
- If absent, there must be an unique placeholder message with the same direction as the element in which the *messageLabel* attribute appears.

An <infault> or <outfault> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more elements that does not belong to the namespace “http://www.w3.org/ns/wsdl”.

The Service Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Service

A Service Component specifies one or more endpoints (IP addresses) at which an implementation of a web service having a specified interface is available.
The XML representation of the Service Component looks like this:

```
<description>
  ...
  <service name="xs:NCName" interface="xs:QName">
    <endpoint name="xs:NCName" binding="xs:QName" address="xs:anyURI"?
              whttp:authenticationScheme="xs:token"??
              whttp:authenticationRealm="xs:string"?? >
      <documentation />*
    </endpoint>+
  </service>
  ...
</description>
```

The <service> element can have the following attributes:

Attribute Name	Required	Description
name	Yes	With the target namespace of the WSDL document, the value of the *name* attribute forms the qualified name which can be used to refer to the service. Must be unique within a Description Component.
interface	Yes	Reference to the (abstract) Interface Component which the service instantiates.

A <service> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- One or more <endpoint> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what a Service Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <service name="reservationService" interface="tns:reservationInterface">

     <endpoint name="reservationEndpoint"
               binding="tns:reservationSOAPBinding"
               address ="http://greath.example.com/2004/reservation"/>

  </service>
</description>
```

The Endpoint Component

References:
http://www.w3.org/TR/2007/REC-wsdl20-20070626/#Endpoint
http://www.w3.org/TR/wsdl20-adjuncts/#http-auth-decl

An Endpoint Component specifies an address at which a web service using the specified binding can be accessed. The interface of the web service is specified by the parent Service Component.

The XML representation of the Endpoint Component, including the HTTP binding extension, looks like this:

```
<description>
  ...
  <service name="xs:NCName" interface="xs:QName">
    <endpoint name="xs:NCName" binding="xs:QName" address="xs:anyURI"?
              whttp:authenticationScheme="xs:token"?
              whttp:authenticationRealm="xs:string"? >
      <documentation />*
    </endpoint>+
  </service>
  ...
</description>
```

The <endpoint> element can have the following attributes:

Attribute Name	Required	Description
name	Yes	With the target namespace of the WSDL document, the value of the *name* attribute forms the qualified name which can be used to refer to the endpoint. Must be unique within the Service Component in which the Endpoint Component appears.
binding	Yes	Reference to the Binding Component that specifies the binding for this endpoint.
address	No	URI specifying the network address at which the service can be accessed.
whttp:authentication Scheme	No	HTTP authentication scheme used for this endpoint. Possible values: “basic”, “digest”
whttp:authentication Realm	No If the whttp:authenticationScheme attribute present, then this attribute is required.	Realm authentication parameter as defined in RFC2617.

The *address* attribute of the <endpoint> element is optional allowing scenarios where an address is not needed or acquired by other means.

An <endpoint> element has the following properties:

- Belongs to the namespace “http://www.w3.org/ns/wsdl”.
- Zero or more <documentation> elements.
- Zero or more namespace qualified elements whose namespace is not “http://www.w3.org/ns/wsdl”.

Here is an example of what an Endpoint Component may look like in an actual WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<description ...>
  ...
  <service name="reservationService" interface="tns:reservationInterface">

     <endpoint name="reservationEndpoint"
               binding="tns:reservationSOAPBinding"
               address ="http://greath.example.com/2004/reservation"/>

  </service>
</description>
```

3.4 UDDI Publish and Inquiry APIs

Describe the basic functions provided by the UDDI Publish and Inquiry APIs to interact with a UDDI business registry.

References:
http://uddi.xml.org/
http://java.sun.com/webservices/jaxr/overview.html

This section described UDDI v2, since this is the latest version supported by the JAXR API, see the above JAXR reference.

UDDI is a web service that uses SOAP 1.1 over HTTP and Document/Literal encoding and the request/response messaging mode. Noteworthy is that UDDI web services allow the use of UTF-8, but not UTF-16, which essentially means that they do not conform with the the BP. Additionally, UDDI does not support the use of the SOAP <Header> element.

The general structure of a UDDI SOAP message is:

```
POST /someVerbHere HTTP/1.1
Host: www.someoperator.org
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <some-uddi-element generic="2.0" xmlns="urn:uddi-org:api_v2">
      ...
    </some-uddi-element>
  </Body>
</Envelope>
```

To create a UDDI SOAP message, replace the <some-uddi-element> element with the, for the operation or reply in question, appropriate UDDI data structure.

There are two APIs for accessing information in a UDDI web service:

- The UDDI Inquiry API
  Used to search and read data in a UDDI registry.
- The UDDI Publishing API
  Used to add, modify and delete data in a UDDI registry.

The UDDI Inquiry API

The UDDI inquiry API can be used to find and retrieve data from the UDDI registry. These operations can be performed by anyone at anytime.

Find Operations

The UDDI inquiry API contains the following find-operations:

Operation	Description
find_business	Finds matching businessEntity entries.
find_relatedBusiness	Finds matching publisherAssertion entries.
find_service	Finds matching businessService entries.
find_binding	Finds matching bindingTemplate entries.
find_tModel	Finds matching tModel entries.

Many find operations retrieve lightweight lists of corresponding data. To retrieve complete data entries, use the get-operations described below.

Find criteria can be:

- identifierBag (identifiers)
- categoryBag (categories)
- name (names)
- tModelBag
  Applies to tModel-s that describes the web service.
- findQualifiers
  Allows for customization of matching criteria of a search (ex: use case-sensitive matching, sorting of the result etc.).

Note that not all find criteria can be used with all operations.

Get Operations

The get operations retrieves zero or more entries of the requested data structure that has an unique identifier matching those supplied in the request.
The UDDI inquiry API contains the following get-operations:

Operation	Description
get_businessDetail	Gets businessEntity entries.
get_businessDetailExt	Gets businessEntityExt entries.
get_serviceDetail	Gets businessService entries.
get_bindingDetail	Gets bindingTemplate entries.
get_tModelDetail	Gets tModel entries.

The UDDI Publishing API

The UDDI publishing API is used to add, change and delete information in an UDDI registry. Unlike the UDDI inquiry API, the publishing API requires access using HTTPS and authentication and authorization.
Every message, except for the login message, must include an authorization token that is issued by the UDDI registry at the beginning of a session. Such a token is unique for the session and valid only during the lifetime of the session.
The UDDI publishing API supports four kinds of operations:

Operation Kind	Description
Authorization Operations	Allows for client authentication, obtaining an authorization token, termination of a session and its authorization token.
Delete Operations	Removal of the primary data structures of the registry.
Get Operations	Retrieval of publisherAssertion entries and summary of registered information.
Save Operations	Adding or updating of the primary data structures of the registry.

Authorization Operations

Authorization operations are used to log in and out of a UDDI registry.

Operation	Description
get_authToken	Log in to the UDDI registry.
discard_authToken	Log out of the UDDI registry.

Save Operations

Save operations are used to add or update information in the UDDI registry.

Operation	Description
save_business	Adds or update one or more businessEntity entries.
save_service	Adds or updates one or more businessService entries.
save_binding	Adds or updates one or more bindingTemplate entries.
save_tModel	Adds or updates one or more tModel entries.
add_publisherAssertions	Adds one or more publisherAssertion entries.
set_publisherAssertions	Updates one or more publisherAssertion entries.

Both parties of a relation must perform complementary additions or modifications of publisherAssertion entrie(s), in order for the entrie(s) to remain visible.

Delete Operations

Delete operations are used to remove information from the UDDI registry.

Operation	Description
delete_business	Deletes one or more businessEntity entries. Also deletes businessService, bindingTemplate and publisherAssertion entries owned and exclusively referred to by the businessEntity entries.
delete_service	Deletes one or more businessService entries. Aslo deletes bindingTemplate entries owned and exclusively referred to by the businessService entries.
delete_binding	Deletes one or more bindingTemplate entries.
delete_tModel	Makes one or more tModel entries invisible to find-operations.
delete_publisherAssertions	Deletes one or more publisherAssertion entries.

Get Operations

Get operations are used to retrieve summary data about data structures published by the client.

Operation	Description
get_assertionStatusReport	Gets a summary of publisherAssertion entries. Whether they are complete or incomplete.
get_publisherAssertions	Gets a list of all publisherAssertion entries, visible or invisible, submitted by the client.
get_registeredInfo	Gets an abbreviated list of businessEntity and tModel entries controlled by the client.

Faults

Errors occurring when interacting with an UDDI registry are reported as SOAP faults containing disposition report in the SOAP fault <detail> element. A disposition report has the following format:

```
    <xsd:element name="dispositionReport" type="uddi:dispositionReport"/>
    <xsd:complexType name="dispositionReport">
        <xsd:sequence>
            <xsd:element ref="uddi:result" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="generic" type="string" use="required"/>
        <xsd:attribute name="operator" type="string" use="required"/>
        <xsd:attribute name="truncated" type="uddi:truncated" use="optional"/>
    </xsd:complexType>

    <xsd:element name="result" type="uddi:result"/>
    <xsd:complexType name="result">
        <xsd:sequence>
            <xsd:element ref="uddi:errInfo" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="keyType" type="uddi:keyType" use="optional"/>
        <xsd:attribute name="errno" type="int" use="required"/>
    </xsd:complexType>
```

Below is an example of a SOAP fault containing a disposition report:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoaporg.org/soap/envelope/">
    <Body>
        <Fault>
            <faultcode>Client</faultcode>
            <faultstring>Client Error</faultstring>
            <detail>
                <dispositionReport
                    generic="2.0"
                    operator="OperatorURI"
                    xmlns="urn:uddi-org:api_v2">
                    <result errno="12312">
                        <errInfo errCode="E_fatalError ">
                            Some error message
                        </errInfo>
                    </result>
                </dispositionReport>
            </detail>
        </Fault>
    </Body>
</Envelope>
```

Some operations, such as discard_authToken, also return a disposition report upon successful completion. In such a case, the <dispositionReport> is a child of the SOAP <Body> element.

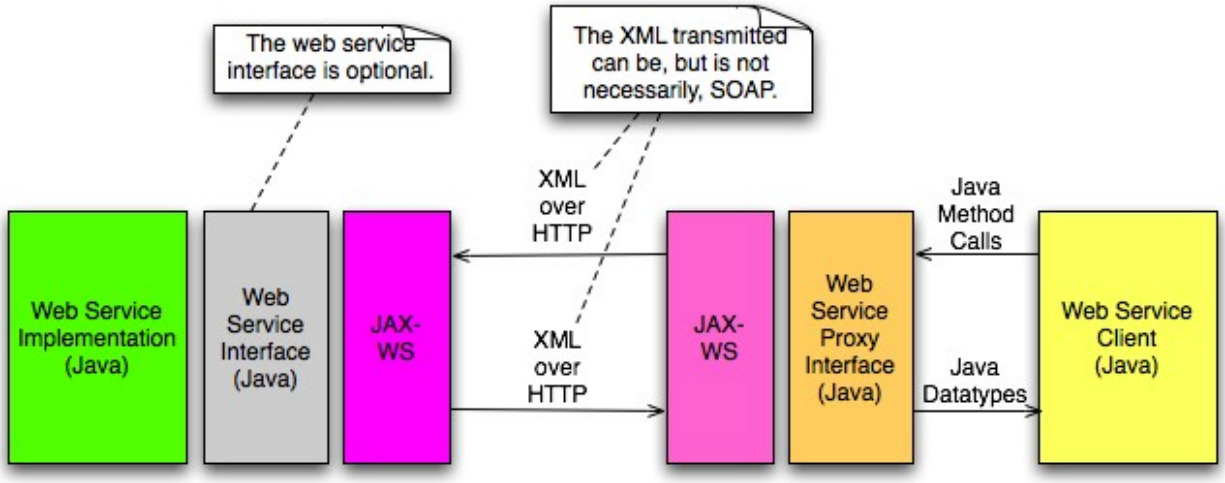
4. JAX-WS

4.1 JAX-WS Technology

Explain JAX-WS technology for building web services and client that communicate using XML

References:
JavaEE 5 Tutorial, chapter 16.
https://jax-ws.dev.java.net/nonav/2.1/docs/UsersGuide.html

The following figure illustrates a typical invocation of a web service that uses JAX-WS at both the client and server side.



A typical client invocation of a web service consists of the following steps:

1. The web service client invokes a Java method in the web service proxy interface, supplying parameters.
2. JAX-WS generates the appropriate SOAP message for the invoked method.
3. JAX-WS maps the parameters of the method call to XML using JAXB and inserts them into the SOAP message.
4. JAX-WS sends the SOAP message over HTTP to the web service server.
5. At the server side, JAX-WS interprets the SOAP message, mapping it to appropriate Java method in the web service implementation.
6. JAX-WS maps the parameters XML data to Java parameters.
7. JAX-WS invokes the Java method in the web service implementation.
8. The web service implementation method processes the request and returns some data.
9. JAX-WS generates the appropriate SOAP response message.
10. JAX-WS maps the resulting data of the Java web service implementation method to XML using JAXB and inserts it into the SOAP message.
11. JAX-WS returns the HTTP response to the client.
12. On the client side, JAX-WS maps the XML data to Java return data.
13. The call to the method in the web service proxy interface returns, supplying the return data.

From this we can see that JAX-WS isolates the programmer from complexity to a very large extent. JAX-WS also provides tools that enables generation of both client and server side artifacts, in order to further reduce the amount of work required to implement a web service and/or a web service client. We will use these tools later when implementing a JAX-WS web service and JAX-WS clients.
Additionally:

- JAX-WS supports both message-oriented web services as well as RPC-oriented web services.
- JAX-WS enables development of both web services and web service clients in the platform independent Java language.
- JAX-WS 2.0 allows for direct sending of XML over HTTP, thus supporting RESTful web services.
- JAX-WS uses technologies defined by the World Wide Web Consortium (W3C), such as HTTP, SOAP and WSDL, which are standard technologies applicable regardless of the programming language or platform used for web services and web service clients.
- JAX-WS supports the WS-I Basic Profile version 1.1, which clarifies SOAP and WSDL specifications and establish a set of best practices that promotes interoperability.
- JAX-WS supplies a layered programming model consisting of two layers:
  - The upper layer, which is the only layer most applications will use, uses annotations extensively which makes it easy to use.
  - The lower layer is more traditional, API-based, and more suitable for special cases.

4.2 Developing JAX-WS Web Services

Given a set of requirements for a Web service, such as transactional needs and security requirements, design and develop Web service applications that use JAX-WS technology

A JAX-WS web service can be implemented either using an EJB 3 endpoint or using a servlet endpoint. It should be noted that both EJB based endpoints and POJO based endpoints will be wrapped by a servlet.
Both servlet endpoints and EJB endpoints can have an object implementing the *WebServiceContext* interface injected using the @Resource annotation. This interface contains the following methods:

Method	Description
MessageContext getMessageContext()	Retrieves the message context of the request being served. The basic message context is of the type *Map<String, Object>*.
Principal getUserPrincipal()	Retrieves the user's principal identifying the sender of the request, or null if there is none.
boolean isUserInRole(String role)	Determines if an authenticated user is in the role having the supplied role name.

Using a *WebServiceContext* object, the endpoint have access to basic security mechanisms of JavaEE. Both servlet-based and EJB-based endpoints can access the *MessageContext* of a web service context which contains, for instance, the following things:

- Servlet request object (type: *javax.servlet.http.HttpServletRequest*)
- Servlet response object (type: *javax.servlet.http.HttpServletResponse*)
- HTTP request headers (type: *java.util.Map*)

One thing that deserves special mentioning is that servlet-based endpoints additionally can access a servlet context (type: *javax.servlet.ServletContext*). For a complete listing on what kinds of objects that can be retrieved from the message context, see the *javax.xml.ws.handler.MessageContext* API documentation.

Servlet Endpoints

Servlet endpoints:

- Have access to basic security mechanisms of JavaEE.
  However, these do not allow for as fine grained control as in the case with an EJB endpoint.
- Does not manage multi-threading issues.
- Does not require an EJB container.
- Have access to *HttpSession* object.
  This object can be used for maintaining short duration client state.

EJB Endpoints

An EJB endpoint is a regular Enterprise Java Bean version 3 and as such provide:

- Transactions
- Security
  More fine grained security, where permissions can be specified for individual methods.
- Interceptors
- Access to timer services
- Dependency injection
- Thread management
  Only one thread at a time is allowed to execute in an EJB endpoint instance.

Summary

Choose an EJB endpoint if:

- An existing stateless EJB needs to be exposed as a web service.
- The business logic that the service uses is in an EJB tier.
  In order for the endpoint and the business logic to reside in the same tier.
- You want transaction and/or security services offered by the EJB container.
- You want to be able to control security at method level.
- You want the container to manage concurrent access to endpoint instance(s).

Choose a servlet endpoint if:

- Less complexity is required.
- Less execution overhead is required.
- You want to be able to access the web service servlet context from the web service.
- You want to be able to access the *HttpSession* object from the web service.
- The business logic that the service uses is in a web tier.
  In order for the endpoint and the business logic to reside in the same tier.
- You need a lightweight container.
  EJB endpoints require an EJB container and a servlet container, servlet endpoints only need a servlet container.
- You are prepared to manage concurrent access by multiple threads to the endpoint.

4.3 The I-Stack

Describe the Integrated Stack (I-Stack) which consists of JAX-WS, JAXB, StAX, SAAJ.

References:
https://jax-ws-architecture-document.dev.java.net/nonav/doc-201/com/sun/xml/ws/package-summary.html

API	Description	Function in the I-Stack
JAX-WS	Java API for XML Web Services	API for creating web services.
JAXB	Java Architecture for XML Binding	Marshaling of Java objects to XML representation and unmarshaling of XML representations to Java objects.
StAX	Streaming API for XML	Reading and writing of XML documents.
SAAJ	SOAP with Attachments API for Java	Facilitates production and consumption of SOAP messages with attachments.

From the above reference:
JAX-WS is the aggregating component of what is called the integrated Stack (I-Stack). The I-Stack consists of:

- JAX-WS
- JAXB
  The databinding component of the stack.
- StAX
  The Streaming XML parser used by the stack.
- SAAJ
  Used for its attachment support with SOAP messages and to allow handler developers to gain access to the SOAP message via a standard interface.
- Fast Infoset
  A binary encoding of XML that can improve performance.

4.4 JAX-WS Development Approaches

Describe and compare JAX-WS development approaches.

There are three development approaches that can be used when developing JAX-WS web services:

- Java first
  Also called “bottom-up”.
- WSDL first
  Also called “top-down” or “contract first”.
- Meet in the middle

Java First

- Write the endpoint class in Java.
- Optionally write the service endpoint interface (SEI).
  This interface is a Java interface exposing the methods of the web service.
- Annotate the endpoint class, or the SEI if it has been written, using JAX-WS annotations.
- Generate the artifacts needed to deploy the web service.
  Many containers will generate such artifacts when the web service is deployed, thus eliminating this step. Automatic artifact generation at deployment time is not mandated by the JAX-WS specifications.
- Package the archive (WAR or JAR) containing the web service.
- Deploy the archive in the appropriate container.

For an example using this approach, please refer to section 4.7 below.

WSDL First

- If needed, write the WSDL document describing the web service.
- Generate the service endpoint interface (SEI) and additional artifacts from the WSDL file.
- Create the service implementation by writing a class that implements the SEI.
- Package the archive (WAR or JAR) containing the web service.
- Deploy the archive in the appropriate container.

For an example using this approach, please refer to section 9.3 below.

Meet in the Middle

The prerequisites for the Meet in the Middle approach are an existing WSDL document and an existing implementation of the service. This approach may be used when a web service with a fixed interface (the WSDL document) is to be created using existing implementation of business logic.

- Generate the service endpoint interface (SEI) and additional artifacts from the WSDL file.
- Create a service implementation class that implements the SEI and delegates the performing of business functionality to the existing implementation.
- Package the archive (WAR or JAR) containing the web service.
- Deploy the archive in the appropriate container.

Summary

Approach	Advantages	Disadvantages
Java First	- Requires less knowledge about WSDL, XML etc. - Strong tool support. - More natural approach to Java programmers.	- Interoperability problems may occur with services/clients defined in other languages. - More sensitive to changes. If the endpoint interface changes, the WSDL most likely will have to change and that might require rewriting clients. - Less control over WSDL creation.
WSDL First	- Better interoperability between different web service technologies (.NET, Java etc). - Less sensitive to changes. - More control.	- Requires more knowledge (about WSDL, XML, WS-I Basic Profile etc).
Meet in the Middle	- This approach is chosen out of necessity, due to the other approaches not being applicable. The advantage is that is is able to solve problems that the other approaches cannot.	- Overly complex.

4.5 JAX-WS Features

Describe the features of JAX-WS including the usage of Java Annotations.

The table below lists and comments on JAX-WS features.

JAX-WS Feature	Comments
Dynamic and Static Clients	Web services can be invoked using both generated classes (static clients) and dynamically created *javax.xml.ws.Service* and *javax.xml.ws.Dispatch* instances.
Invocation with Java Interface Proxies	Service Endpoint Interfaces are used to create client proxies that can be used to invoke a web service.
Invocation with XML/ XML Service Providers	Serialization can be bypassed entirely, allowing for interaction with services using raw XML data.
Message Context	JAX-WS passes a message context, which can contain properties, such as username and password for HTTP authentication, along with an XML message from the client, through any handlers on both the client and server side, to the server (or reverse direction for response messages).
Handler Framework	Handlers is similar to filters or interceptors for web service requests and response that allows for pre- and post-processing of messages.
SOAP Binding	JAX-WS provides a SOAP binding for SOAP message processing.
HTTP Binding	JAX-WS provides a XML/HTTP binding which, for instance, allows for implementation of RESTful web services or other services which do not use SOAP.
Converting Exceptions to SOAP Faults	JAX-WS can map Java exceptions to SOAP faults and the reverse. This eliminates the need to write code that converts service exceptions to SOAP faults and the reverse.
Asynchronous Invocation	JAX-WS supports asynchronous invocation of web services. Two models for response notification are provided: Callback and polling.
One-Way Operations	JAX-WS supports one-way, “fire and forget”, operations which allows for loosely coupled applications based on messaging.
Client Side Thread Management	A *javax.xml.ws.Service* instance allows for setting of a *java.util.concurrent.Executor*, which allows for custom thread management.

Pseudo Reference Passing	Using the *javax.xml.ws.Holder<T>* class.
WSDL Styles - Support for RPC/Literal and Document/Literal Wrapped	JAX-WS supports two WS-I compliant WSDL styles: RPC/Literal and Document/Literal wrapped.
Java/WSDL Mapping	When developing a Java web service, two options are: - Java First: WSDL is generated from Java code. - WSDL First: Java code is generated from WSDL.
Static WSDL	JAX-WS allows for bypassing of the automatic WSDL generation, instead using a static WSDL document.
XML Catalogs	JAX-WS supports OASIS XML Catalogs 1.1, enabling mapping of XML schema URLs to local copies of the XML schemas.
Run-Time Endpoint Publishing (JavaSE only)	Enables runtime publishing of a web service endpoint at runtime using the *javax.xml.ws.Endpoint*.

Annotations

References:
JAX-WS 2.1 Specification, section 6.5, chapter 7.
http://www.w3.org/TR/ws-addr-wsdl/
http://www.w3.org/TR/soap12-mtom/

Annotations play an important role in JAX-WS. They are used to map Java to WSDL and XML schema and, to some extent, the reverse. They are also used to control how the JAX-WS runtime processes and responds to web service invocations. The following annotations are available:

Annotation	Purpose
Web Services Metadata Annotations	
WebService	Mark the annotated endpoint implementation class as implementing a web service or mark the annotated service endpoint interface as defining a web service interface.
WebMethod	Expose a method as a web service operation or exclude a method from being exposed.
OneWay	Mark a method as a one-way web service operation.
WebParam	Customize the mapping of one method parameter to a WSDL message part or XML element.
WebResult	Customize the mapping of the return value of a method to a WSDL message part or XML element.
HandlerChain	Specify an externally defined handler chain.
SOAPBinding	Specify the messaging style(DOCUMENT or RPC), encoding (only LITERAL available) and parameter style (BARE or WRAPPED) of an endpoint. Default is DOCUMENT/LITERAL, WRAPPED.
JAX-WS Annotations	
BindingType	Specifies the binding (HTTP, SOAP 1.1, SOAP 1.2 etc.) to use when publishing an endpoint implemented by the annotated endpoint class.
RequestWrapper	Specifies the JAXB generated request wrapper bean and corresponding XML element for marshaling and unmarshaling the wrapper bean. - Local name of the XML schema element representing the wrapper. - Namespace of the XML schema element representing the wrapper. - Name of the Java class representing the wrapper.
ResponseWrapper	Specifies the JAXB generated response wrapper bean and corresponding XML element for marshaling and unmarshaling the wrapper bean. - Local name of the XML schema element representing the wrapper. - Namespace of the XML schema element representing the wrapper. - Name of the Java class representing the wrapper.
ServiceMode	Used in conjunction with the *javax.xml.ws.Provider<T>* interface to indicate that the endpoint wants to access protocol messages or protocol message payloads.
WebEndpoint	Associates the annotated *getPortName()* method in a generated class implementing the *javax.xml.ws.Service* interface with a specified <wsdl:port> element. Such a class provides the client view of a web service.
WebFault	Specifies mapping between WSDL fault and Java exception and/or service-specific exceptions and WSDL faults.
WebServiceFeature	Meta annotation used by JAX-WS to identify other annotations as web service features. Also see section on web service feature annotations below!

Annotation	Purpose
WebServiceClient	Annotates a generated web service class implementing the *javax.xml.ws.Service* interface. Associates the class with a web service by specifying the location of the WSDL document, the name of the <wsdl:service> element and the namespace of the <wsdl:service> element.
WebServiceProvider	Used in endpoint classes implementing the *javax.xml.ws.Provider<T>* to associate the class with a <wsdl:service> element and a <wsdl:port> element in specified WSDL document. Such an endpoint works directly with (SOAP) messages or (SOAP) message payloads. A class can either be annotated with the WebService or the WebServiceProvider annotation, but not both.
WebServiceRef	Used to define a reference to a web service and, optionally, an injection target in which the reference will be injected.
WebServiceRefs	Used to define multiple references to web services. Used in conjunction with the WebServiceRef annotation above.
Action	Specifies the WS-Addressing Action values for input, output and fault messages of the WSDL operation (annotated method). See http://www.w3.org/TR/ws-addr-wsdl/#actioninwsdl for more information on the Action message addressing property.
FaultAction	Used inside an Action annotation to specify a WS-Addressing Action value for a fault message which applies to a certain, specified, exception of the WSDL operation.
JAXB Annotations	
XmlRootElement	Specify the XML schema local name and namespace of the global XML element in a WSDL document which will represent the top level class when mapping between XML and instances of the class.
XmlAccessorType	Specifies whether fields and/or properties of a class will be serialized or not when producing an XML representation of an instance of the class.
XmlType	Specifies the XML schema local name and namespace of the XML type representing instances of the class. Also specifies order of the XML schema elements to which the properties of instance of the annotated class are mapped.
XmlElement	Specifies the XML schema local name and namespace of the local element in the XML schema complex type, which represents the class in which the property is located, that will represent the annotated property.
XmlSeeAlso	Instruct JAXB to bind additional, specified, classes when binding the annotated class.
Common Annotations	
Resource	Annotate a field or method of a JAX-WS endpoint class that will be injected with a *WebServiceContext* object.
PostConstruct	Specifies a method that will be invoked after dependency injection has been performed and before the instance of the class is put into service, that is, an initialization method.
PreDestroy	Specifies a method that will be invoked immediately before the instance of the class is taken out of service by the container, that is, a clean-up method.
Web Service Feature Annotations	
Addressing	Specifies whether or not WS-addressing is used for the annotated endpoint implementation class. Also specifies whether or not addressing headers are required to be present in incoming messages.
MTOM	Specifies whether or not MTOM is enabled for the annotated endpoint implementation class.
RespectBinding	Specifies whether or not the annotated endpoint implementation class must respect the <wsdl:binding> associated with the endpoint.

Additional Features

The JAX-WS specification introduces the following Additional Features, which I am not sure are to be included in this section. For the sake of completeness, they have been included.

JAX-WS 2.1 introduces the notion of features. A feature is associated with certain functionality or behaviour. The three standard features introduced are:

- AddressingFeature
- MTOMFeature
- RespectBindingFeature

Features can be enabled or disabled. Each feature is represented by a class that inherits from the abstract class *WebServiceFeature*.

AddressingFeature

From the JAX-WS API:
This feature represents the use of WS-Addressing with either the SOAP 1.1/HTTP or SOAP 1.2/HTTP binding. Using this feature with any other binding is not required.
Enabling this feature on the server will result in the <wsaw:UsingAddressing> element being added to the <wsdl:Binding> for the endpoint and in the runtime being capable of responding to WS-Addressing headers.
Enabling this feature on the client will cause the JAX-WS runtime to include WS-Addressing headers in SOAP messages.

From the Web Services Addressing 1.0 specification document:

> Web services addressing provides transport-neutral mechanisms to address web services and messages.

MTOMFeature

The MTOM feature is used to enable or disable transport optimization of SOAP messages as well as setting the threshold used to determine when binary data should be encoded.

RespectBindingFeature

From the JAX-WS API:

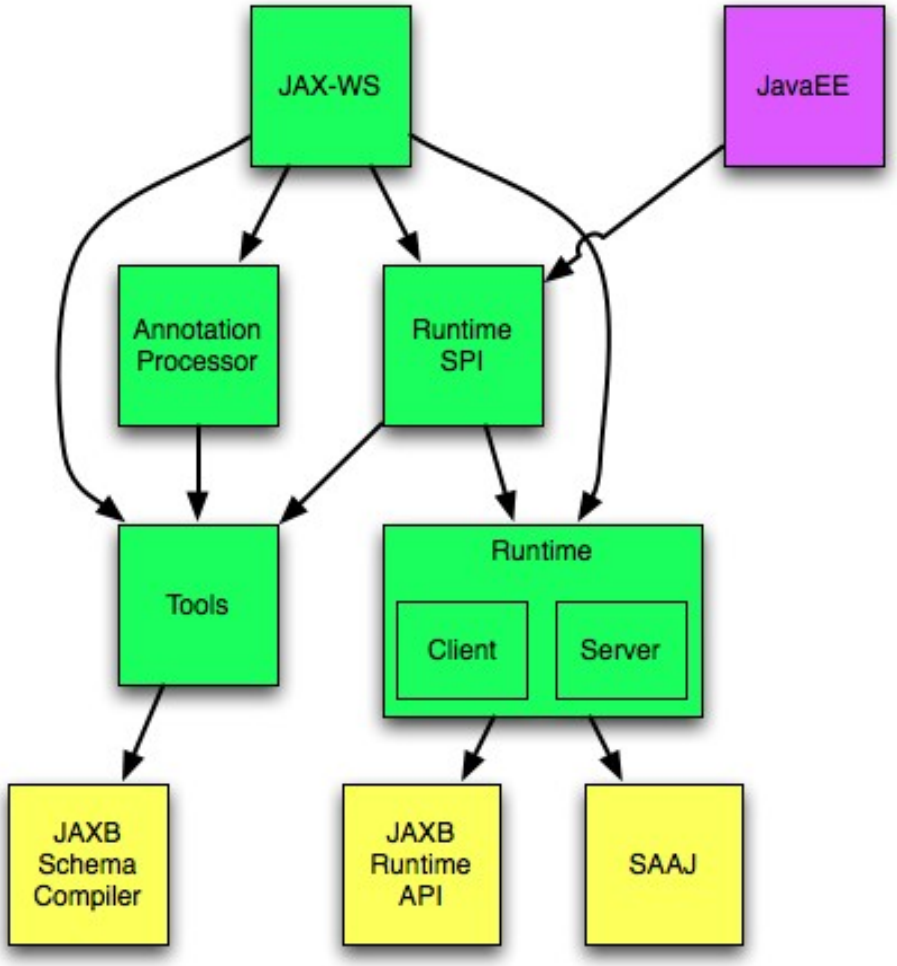
> This feature clarifies the use of the <wsdl:binding> in a JAX-WS runtime.
> This feature is only useful with web services that have an associated WSDL. Enabling this feature requires that a JAX-WS implementation inspect the <wsdl:binding> for an endpoint at runtime to make sure that all <wsdl:extensions> that have the required attribute set to true are understood and are being used.

4.6 JAX-WS Architecture

Describe the architecture of JAX-WS including the Tools SPI that define the contract between JAX-WS tools and Java EE.

References:
https://jax-ws-architecture-document.dev.java.net/

The following picture shows the major components of JAX-WS in green, components of external APIs used by JAX-WS in yellow and a JavaEE client module in purple.



JAX-WS architecture; the main components (in green).

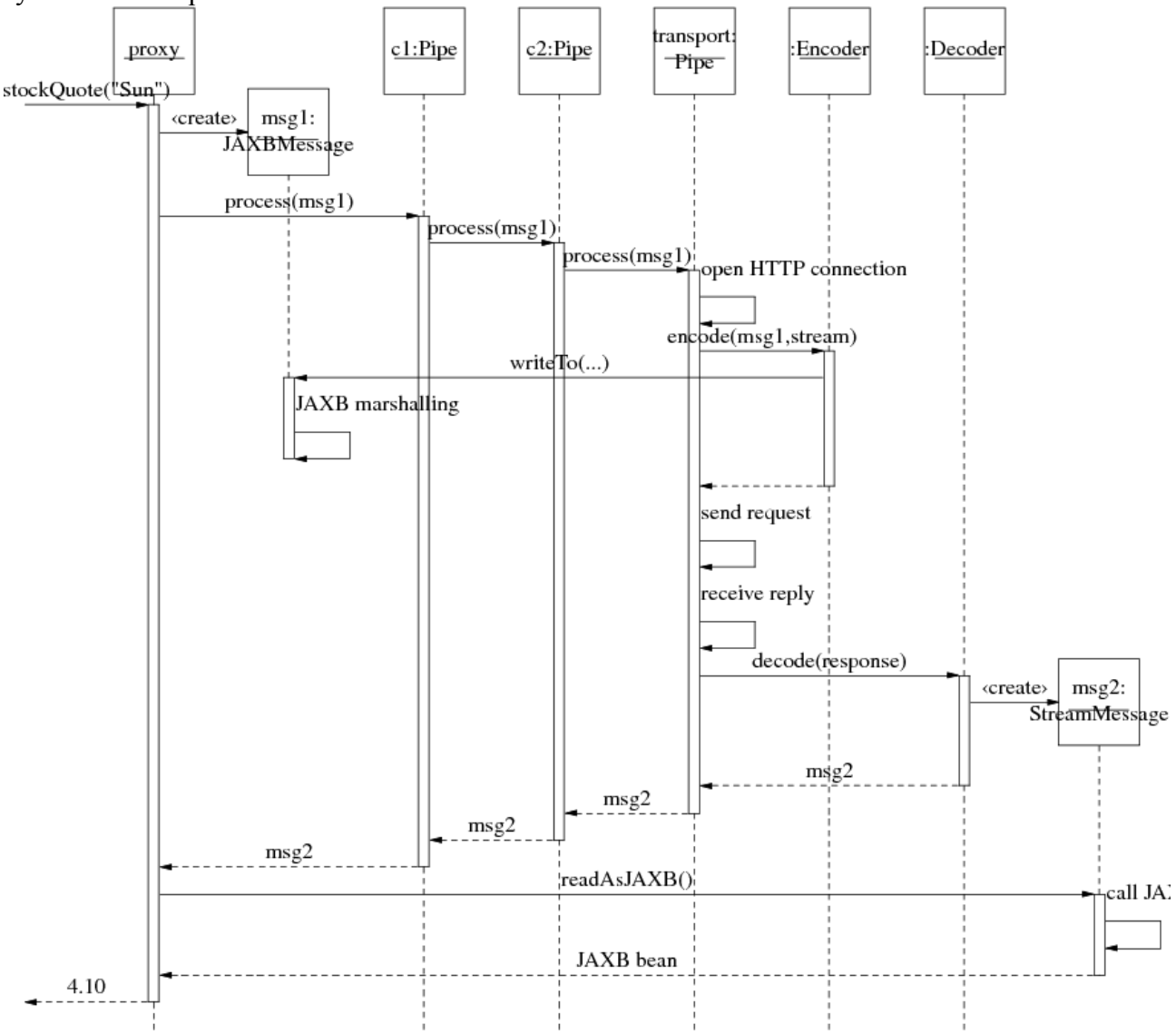
Additionally, not shown in this picture, JAX-WS uses the APT annotation processing tool from JavaSE.

Component	Description
JAX-WS	Top level component.
Annotation Processor	Extends the JavaSE annotation processing tool by supplying processing of javax.jws.*, javax.jws.soap.*, and javax.xml.ws.* annotations.
Runtime SPI	Defines the contract between the JAX-WS runtime and Java EE.
Tools	JAX-WS supplies the following three tools: - APT (annotation processing tool) Compiles Java source files and generates any additional classes needed to make an @WebService annotated class a Web service. - wsgen Processes a compiled @WebService annotated class and generates the necessary classes to make it a Web service. - wsimport Generates, from WSDL documents, either client or server-side service endpoint interfaces.
Runtime	Contains the core web services framework. Divided into a client and a server side portion, see below for more information.

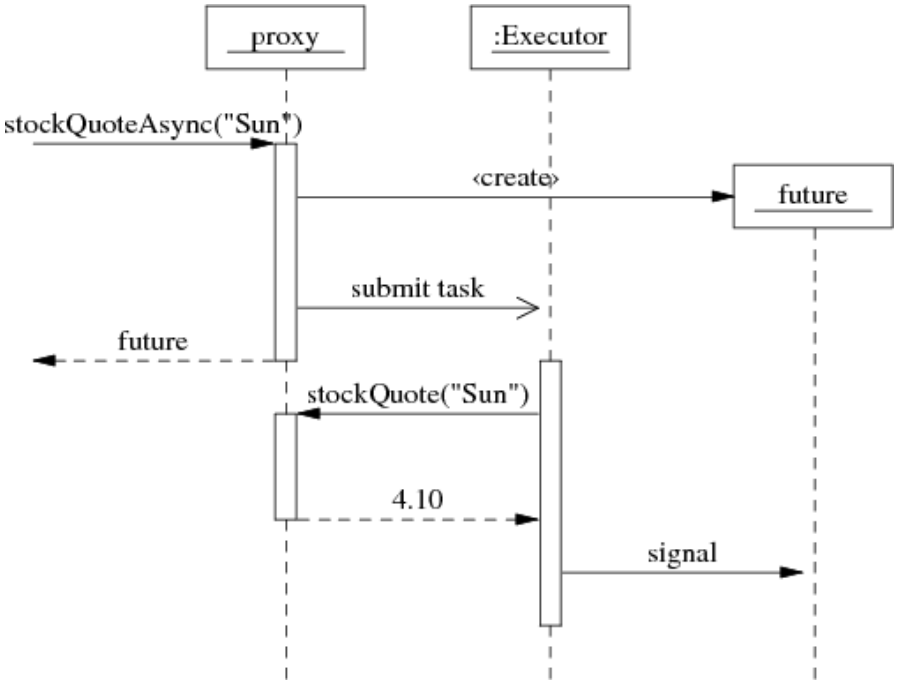
Client Side JAX-WS Runtime

References:
https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/client/package-summary.html

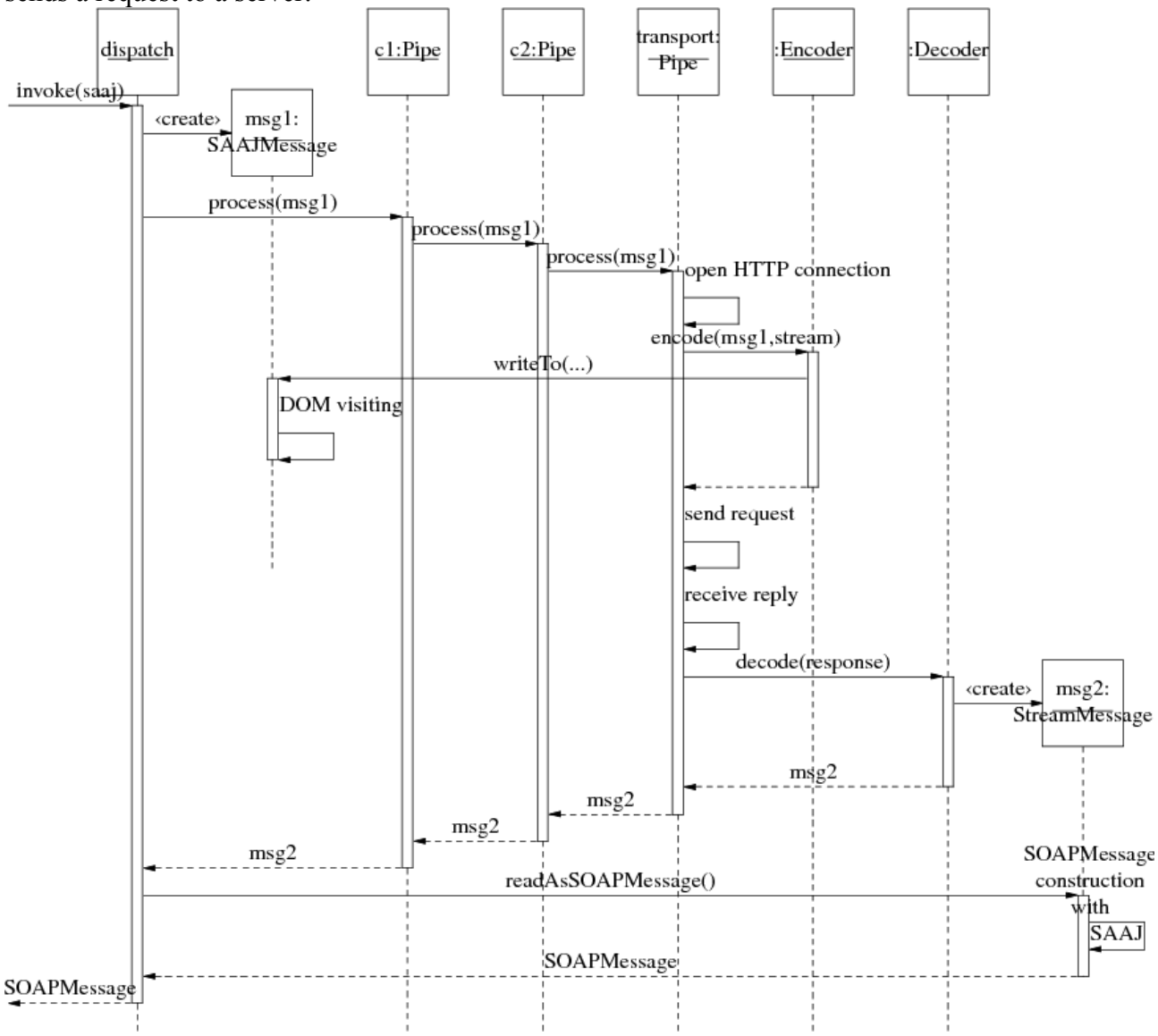
The following diagram describes the basic process of a JAX-WS web service client sending a synchronous request to a server:



The following diagram describes how a JAX-WS client sends an asynchronous web service request to a server:



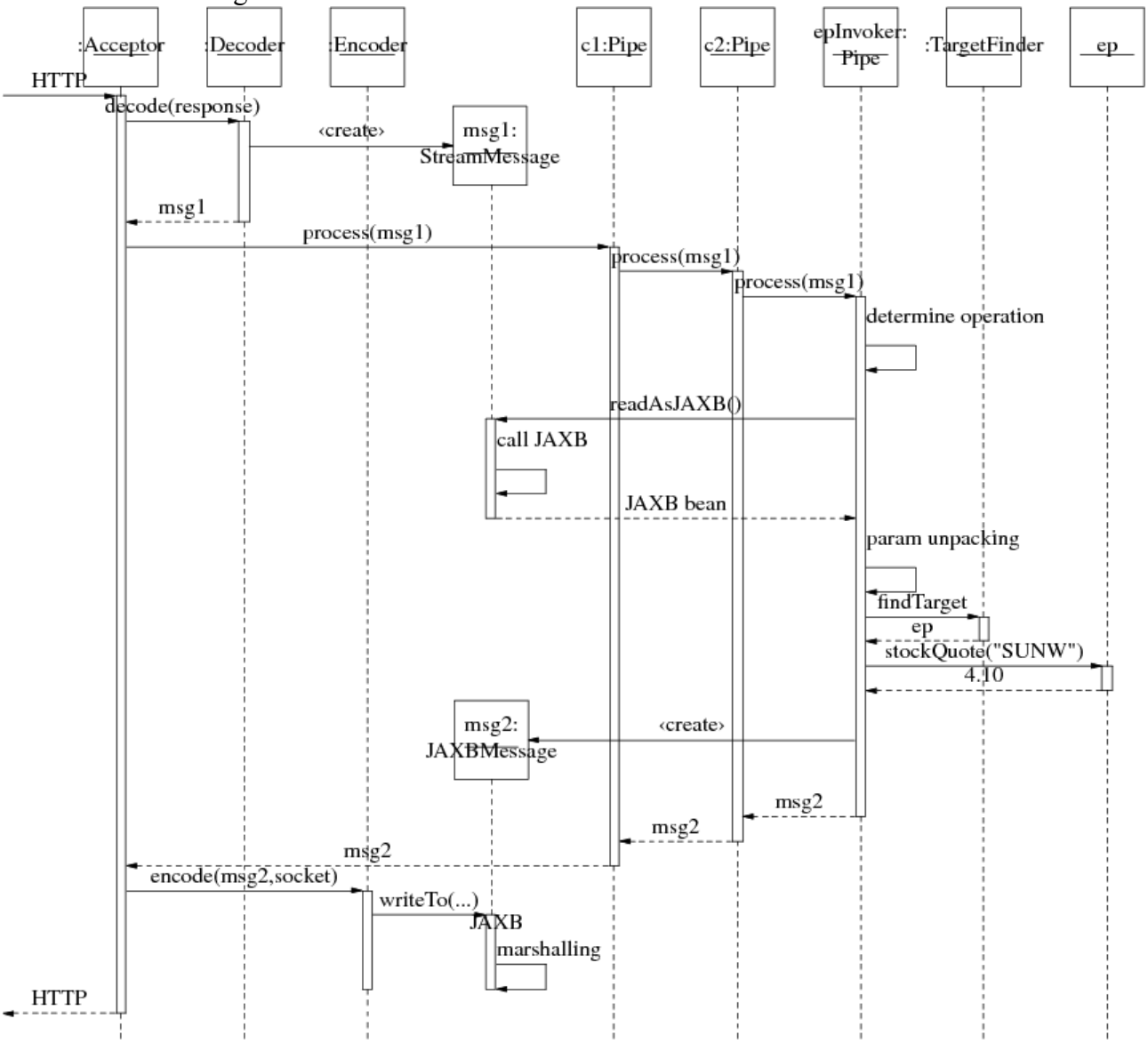
Finally, the following diagram describes how a JAX-WS client working at the XML message level sends a request to a server:



Server Side JAX-WS Runtime

References:
https://jax-ws-architecture-document.dev.java.net/nonav/doc/com/sun/xml/ws/server/package-summary.html

The following diagram describes the basic process of a JAX-WS server receiving and responding to a web service message:



JAX-WS Tools SPI

References:
http://www.java2s.com/Open-Source/Java-Document/6.0-JDK-Modules/jax-ws-tools/com.sun.tools.ws.spi.htm

The JAX-WS Tools SPI provides a means to implement custom behaviour of the wsgen and wsimport tools. It consists of two classes; the abstract *WSToolsObjectFactory* and the concrete class *WSToolsObjectFactoryImpl*. both located in the *com.sun.tools.ws.spi* package.
The *WSToolsObjectFactory* defines the properties of a factory used to produce JAX-WS tools related objects and contains the following methods:

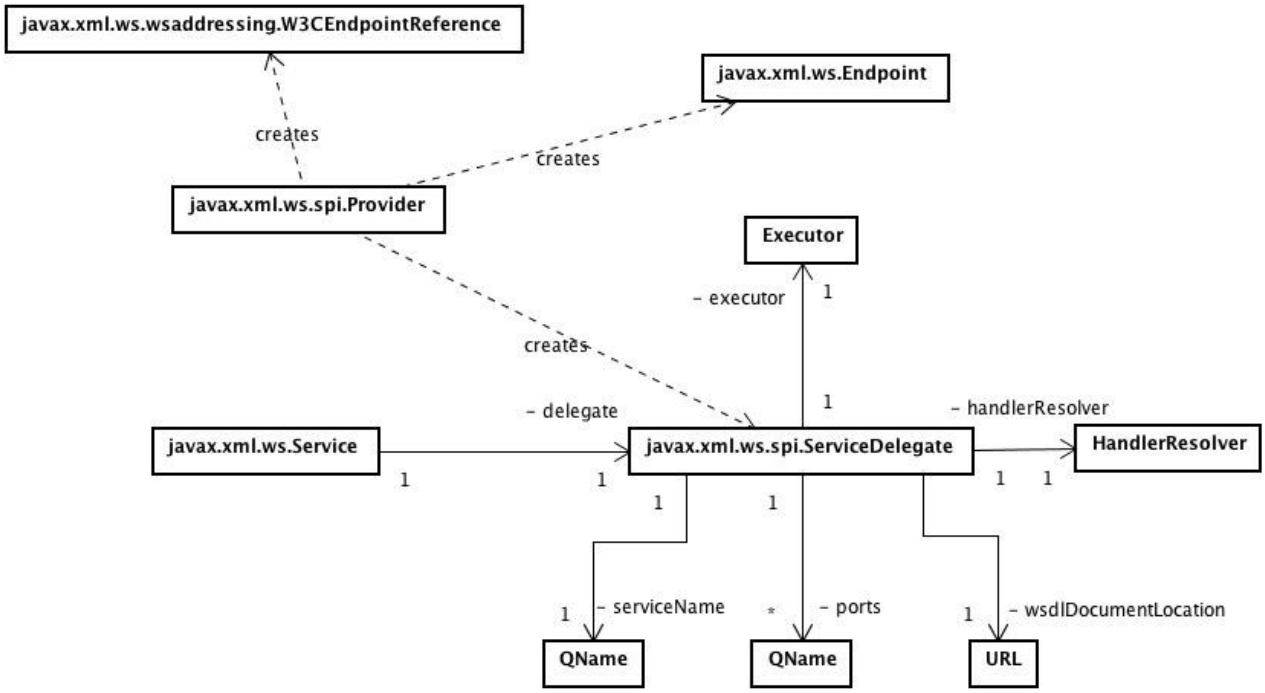
Method Signature	Description
public static WSToolsObjectFactory newInstance()	Obtain an instance of the factory.
abstract public boolean wsgen(OutputStream logStream, Container container, String[] args)	Invokes wsgen on an endpoint implementation, and generates the necessary artifacts like wrapper, exception bean classes etc.
public boolean wsgen(OutputStream logStream, String[] args)	Invokes wsgen on an endpoint implementation, and generates the necessary artifacts like wrapper, exception bean classes etc.
abstract public boolean wsimport(OutputStream logStream, Container container, String[] args)	Invokes wsimport on a WSDL URL argument, and generates the necessary portable artifacts like Service Endpoint Interface, Service, Bean classes etc.
public boolean wsimport(OutputStream logStream, String[] args)	Invokes wsimport on a WSDL URL argument, and generates the necessary portable artifacts like Service Endpoint Interface, Service, Bean classes etc.

The *WSToolsObjectFactoryImpl* class inherits from the *WSToolsObjectFactory* class and provides implementations of the abstract methods for the default behaviour of the tools.

JAX-WS Provider SPI

References:
JAX-WS 2.1 Specification, section 6.2

The Provider SPI allows an application to customize the JAX-WS implementation by defining the properties of the factory that creates endpoints, creates service delegate objects and publishes endpoints. Any customization is totally transparent and requires no changes in the application code. The following diagram gives an overview of relationships between the class in the Provider SPI and other classes in the JAX-WS implementation:



All methods in the *javax.xml.ws.Service* class, except for two *create*-methods that create instances of the *Service* class, are also present in the *javax.xml.ws.spi.ServiceDelegate*. The *Service* class delegates work to the *ServiceDelegate* class.
Subclasses of the abstract *javax.xml.ws.spi.Provider* class are responsible for creating the following types of objects specific to the JAX-WS implementation in question:

- javax.xml.ws.Endpoint
- javax.xml.ws.ServiceDelegate
- javax.xml.ws.wsaddressing.W3CEndpointReference

The abstract *javax.xml.ws.spi.Provider* class contains the following methods:

Method Signature	Description
abstract Endpoint createAndPublishEndpoint(String address, Object implementor)	Creates and publishes an endpoint with the supplied address and protocol to use, both specified by one URI) that uses the supplied implementation object.
abstract Endpoint createEndpoint(String bindingUri, Object implementor)	Creates an endpoint that uses the binding (e.g. SOAP/HTTP) specified by the supplied binding URI and that uses the supplied implementation object.
abstract ServiceDelegate createServiceDelegate(URL wsdlLocation, QName serviceName, Class serviceClass)	Creates a service delegate object using the supplied location of the WSDL document describing the service, the supplied qualified name of the service and the supplied class which must be an instance of *javax.xml.ws.ServiceDelegate* or a subclass thereof.
abstract W3CEndpointReference createW3CEndpointReference(String address, .QName serviceName, QName portName, List<org.w3c.dom.Element> metadata, String wsdlDocumentLocation, List<org.w3c.dom.Element> referenceParameters)	Creates a W3CEndpointReference with the supplied properties.
abstract <T> T getPort(EndpointReference endpointReference, Class<T> serviceEndpointInterface, WebServiceFeature... features)	Creates a proxy that supports the supplied service endpoint interface and which invokes the supplied endpoint reference. The proxy will be configured using the supplied web service features.
static Provider provider()	Creates a new Provider object. Documentation of this method provides description on how to plug in a custom provider implementation.
protected Provider()	Creates a new instance of the Provider class.

For a description on how a provider is chosen, please refer to section 6.2.1 in the JAX-WS 2.1 Specification.

JAX-WS ServiceDelegate SPI

References:
JAX-WS 2.1 Specification, section 6.3
JAX-WS 2.1 API

Please refer to the diagram in the above section on the JAX-WS Provider SPI to see the relationships between the *ServiceDelegate* class and other classes in JAX-WS.
A service delegate is used internally by a service, in order to allow for customization of the JAX-WS implementation. The abstract *javax.xml.ws.spi.ServiceDelegate* class contains the following methods:

Method Signature	Description
protected ServiceDelegate()	Constructor.
abstract void addPort(QName portName, String bindingId, String endpointAddress)	Creates a new port for the service.
abstract <T> Dispatch<T> createDispatch(EndpointReference endpointReference, Class<T> type, Service.Mode mode, WebServiceFeature... features)	Creates a Dispatch instance that invokes the endpoint specified by the supplied endpoint reference. The Dispatch instance will be configured using the supplied web service features.
abstract Dispatch<Object> createDispatch(EndpointReference endpointReference, JAXBContext context, Service.Mode mode, WebServiceFeature... features)	Creates a Dispatch instance that invokes the endpoint specified by the supplied endpoint reference. The Dispatch instance will be configured using the supplied web service features. Marshaling and unmarshaling of messages or message payloads will be performed by supplied JAXB context.
public abstract <T> Dispatch<T> createDispatch(QName portName, Class<T> type, Service.Mode mode)	Creates a Dispatch instance that allows for dynamic invocation of service endpoint operations on any object that the user chooses.
abstract <T> Dispatch<T> createDispatch(QName portName, Class<T> type, Service.Mode mode, WebServiceFeature... features)	Creates a Dispatch instance that allows for dynamic invocation of service endpoint operations on any object that the user chooses. The Dispatch instance will be configured using the supplied web service features.
abstract Dispatch<Object> createDispatch(QName portName, JAXBContext context, Service.Mode mode)	Creates a Dispatch instance that allows for dynamic invocation of service endpoint operations on any object that the user chooses. Uses the supplied JAXB context to marshal and unmarshal messages and message payloads.
abstract Dispatch<Object> createDispatch(QName portName, JAXBContext context, Service.Mode mode, WebServiceFeature... features)	Creates a Dispatch instance that allows for dynamic invocation of service endpoint operations on any object that the user chooses. Uses the supplied JAXB context to marshal and unmarshal messages and message payloads. The Dispatch instance will be configured using the supplied web service features.
abstract Executor getExecutor()	Retrieve the executor used for asynchronous invocations that requires callbacks for the Service object of the delegate.
abstract HandlerResolver getHandlerResolver()	Retrieves the configured handler resolver which, for each proxy or dispatch instance created, retrieves the handler chain that will be set for those objects.

Method Signature	Description
abstract <T> T getPort(Class<T> serviceEndpointInterface)	Retrieves a configured proxy that supports the supplied service endpoint interface. JAX-WS runtime selects the port and binding.
abstract <T> T getPort(EndpointReference endpointReference, Class<T> serviceEndpointInterface, WebServiceFeature... features)	Retrieves a proxy that supports the supplied service endpoint interface and which invokes the supplied endpoint reference. The proxy will be configured using the supplied web service features.
abstract <T> T getPort(QName portName, Class<T> serviceEndpointInterface)	Retrieves a proxy that supports the supplied service endpoint interface for the port in the WSDL service description that has the supplied name.
abstract <T> T getPort(QName portName, Class<T> serviceEndpointInterface, WebServiceFeature... features)	Retrieves a proxy that supports the supplied service endpoint interface for the port in the WSDL service description that has the supplied name. The proxy will be configured using the supplied web service features.
abstract Iterator<QName> getPorts()	Retrieves an iterator that iterates over the qualified names of the service endpoints available in the service.
abstract QName getServiceName()	Retrieves the name of the service.
abstract URL getWSDLDocumentLocation()	Retrieves the location of the WSDL document of the service.
abstract void setExecutor(Executor executor)	Sets the executor used for asynchronous invocations that requires callbacks for the Service object of the delegate.
abstract void setHandlerResolver(HandlerResolver handlerResolver)	Sets the configured handler resolver which, for each proxy or dispatch instance created, retrieves the handler chain that will be set for those objects.

4.7 Creating Web Services with JAX-WS

Describe creating a Web Service using JAX-WS.

References:
Java Web Services Tutorial 2.0, part 1

Developing the most basic form of a JAX-WS web service consists of the following steps:

- Write a Java class that contains the implementation of the service method(s).
- Annotate the above class with JAX-WS annotation(s).
- Compile, package and deploy the web service.

In the above case the web service container will generate the WSDL document and any additional artifacts of the web service.
The procedure of developing a JAX-WS web service when using the JAX-WS wsgen tool is as follows:

- Write a Java class that contains the implementation of the service method(s).
- Annotate the above class with JAX-WS annotation(s).
- Using the JAX-WS wsgen tool, generate additional artifacts of the web service, such as WSDL document and wrapper beans for the request and response data.
- Insert the address of the web service in the WSDL document.
- Add annotations (@RequestWrapper and @ResponseWrapper) to appropriate methods in order for the request and response wrapper classes to be used.
- Compile, package and deploy the web service.

Requirements of a JAX-WS Endpoint

Regardless of the procedure used, there are certain requirements on JAX-WS endpoint class:

- It must be annotated by either the @WebService or the @WebServiceProvider annotation.
- It must not be declared final or be abstract.
- Must have a default public constructor or no constructor.
- It must not define the *finalize* method.
- It may use the @PostConstruct and @PreDestroy annotations on lifecycle callback methods.
- It may specify a service endpoint interface using the *endpointInterface* element in the @WebService annotation, but is not required to do this. If none specified, then one will be implicitly defined.
- Web service methods must be public and must not be final or static.
- Web service methods must be annotated with the @WebMethod annotation, except in the case when all the elements of the @WebMethod annotation have the default values.
- Web service methods must have JAXB compatible parameters and return types.

Calculator Web Service Example (SEI)

In this section a calculator web service will be created using the second approach described above, having wsgen generate web service artifacts prior to deployment. The web service will be developed as a dynamic web project in Eclipse and deployed to GlassFish v2.
The service will have a service endpoint interface (SEI), consisting of the methods in the service implementation class we choose to expose as web service operations. The SEI will, if not explicitly supplied, be generated by the JAX-WS runtime.
Compare this to approach used in the second example below, where the class supplying the functionality of the web service implements the Provider<T> interface and no SEI exists.

Service Implementation Class

The service implementation class contains only two methods; one initialization method and one method that will be exposed as a service. Since we are going to generate the WSDL document and request and response wrapper classes using the wsgen tool, additional annotations have been introduced. Explanation follows below.

```
package com.ivan;

/**
 * Calculator service implementation class.
 */
@WebService(
    name = "Calculator",
    serviceName = "CalculatorService",
    targetNamespace = "http://www.ivan.com/calculator",
    wsdlLocation="CalculatorService.wsdl")
@SOAPBinding(
    parameterStyle=SOAPBinding.ParameterStyle.WRAPPED,
    style=SOAPBinding.Style.DOCUMENT,
    use=SOAPBinding.Use.LITERAL)
public class Calculator
{
    /* Instance variable(s): */
    @Resource private WebServiceContext mWSContext;

    /**
     * Initializes the web service.
     */
    @PostConstruct
    @WebMethod(exclude = true)
    public void init()
    {
        System.out.println("Web service initialized, service context: " + mWSContext);
    }

    /**
     * Adds the supplied numbers.
     *
     * @param inNumber1 First number to add.
     * @param inNumber2 Second number to add.
     * @return Sum of the two numbers.
     */
    @WebMethod(operationName = "addNumbers", action = "urn:Add")
    @ResponseWrapper(
        className = "com.ivan.javax.AddResponse",
        localName = "addNumbersResponse",
        targetNamespace = "http://www.ivan.com/calculator")
    @RequestWrapper(
        className="com.ivan.javax.Add",
        localName="addNumbers",
        targetNamespace="http://www.ivan.com/calculator")
    public int add(final int inNumber1, final int inNumber2)
    {
        System.out.println("Adding " + inNumber1 + " and " + inNumber2);
        return inNumber1 + inNumber2;
    }
}
```

The @WebService annotation tells JAX-WS that this class contains methods that are to be exposed as web services. In addition it contains the following optional elements:

- name = "Calculator"
  Name of the web service. With WSDL 1.1, this will map to the name of the <wsdl:portType> element: <portType name="Calculator">
- serviceName = "CalculatorService"
  Service name of the web service. With WSDL 1.1 this will map to the <wsdl:service> element: <service name="CalculatorService">
- targetNamespace = "http://www.ivan.com/calculator"
  Specifies namespace of <wsdl:portType> and/or <wsdl:service> elements in the associated WSDL 1.1 document.
- wsdlLocation="CalculatorService.wsdl"
  Specifies the location of the WSDL document describing the web service. If none supplied, then GlassFish will generate a WSDL document for us.

The @SOAPBinding annotation specifies the following things about the web service:

- Messaging style:
  style=SOAPBinding.Style.DOCUMENT
- Encoding:
  use=SOAPBinding.Use.LITERAL
- Parameter style:
  parameterStyle=SOAPBinding.ParameterStyle.WRAPPED

The above values are all default values of the SOAPBinding annotation and the annotation could thus have been omitted. Note that the wrapped parameter style is required since there are more than one parameter in the method that is to be exposed as a web service.

Further down, there is an instance variable annotated with the @Resource annotation.

```
...
@Resource private WebServiceContext mWSContext;
...
```

This enables the web service class to have a web service context injected.

The first method in the class is the *init()* method.

```
...
    @PostConstruct
    @WebMethod(exclude = true)
    public void init()
...
```

The @PostConstruct marks a method that is to be invoked after dependency injection has been done on the web service class, but before the web service is taken into use.
The @WebMethod annotation is, in this case, used to exclude the *init()* method from being exposed as a web service method.

Finally, there is the *add(int, int)* method, which is the method we want to expose as a web service.

```
...
    @WebMethod(operationName = "addNumbers", action = "urn:Add")
    @ResponseWrapper(
        className = "com.ivan.javax.AddResponse",
        localName = "addNumbersResponse",
        targetNamespace = "http://www.ivan.com/calculator")
    @RequestWrapper(
        className="com.ivan.javax.Add",
        localName="addNumbers",
        targetNamespace="http://www.ivan.com/calculator")
    public int add(final int inNumber1, final int inNumber2)
...
```

The @WebMethod is, in this case, used to mark a method as to be exposed as a web service operation. In this example, it specifies the following things about the operation:

- operationName=”addNumbers”
  When used with WSDL 1.1, maps to the name of the <wsdl:operation> element in the <wsdl:portType>: <operation name="addNumbers">
- action=”urn:Add”
  Specifies the SOAPAction for the operation in question:
  <soap:operation soapAction="urn:Add" />

The @ResponseWrapper and @RequestWrapper annotations specifies the following about the response and request of the operation:

- className = "com.ivan.javax.AddResponse"
  The name of the Java class implementing the JAXB bean that will be used to hold the response/request data. In this simple example we will never actually come across instances of these classes but they will still be used internally, as we will see.
- localName="addNumbersResponse"
  The name of the XML schema element in which the request/response will be wrapped.
  <xs:element name="addNumbersResponse" type="tns:addNumbersResponse"/>
- targetNamespace="http://www.ivan.com/calculator"
  The namespace of the XML schema in which the element wrapping the request/response is found.

Generating Web Service Artifacts

In this step we will use the wsgen command that comes with either JavaSE 6 or GlassFish to generate additional artifacts for the web service, such as the WSDL document and the JAXB wrapper classes for the request and response of the web service operation.
In order to do this, the following Ant script will be used:

```
<?xml version="1.0"?>

<project default="main" basedir="../../">

      <!-- Root of classpath in which the web service class can be found. -->
      <property name="class-dir" value="${basedir}/build/classes/"/>
      <!-- Default output directory for artifacts generated by wsgen. -->
      <property name="wsgen-outdir" value="${basedir}/build/wsgen-output/"/>
      <!-- Directory to which WSDL document will be written by wsgen. -->
      <property name="wsdl-outdir" value="${basedir}/WebContent/WEB-INF/wsdl/"/>
      <!-- Directory to which generated sourcefiles will be written by wsgen. -->
      <property name="src-outdir" value="${basedir}/src/"/>
      <!-- Directory to which generated classfiles will be written by wsgen. -->
      <property name="gen-classdir" value="${wsgen-outdir}/com/"/>
      <!-- Absolute path of the wsgen command. -->
      <property name="wsgen-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsgen"/>

      <echo message="calling the web services generation task wsgen" />
      <target name="main">
            <exec executable="${wsgen-cmd}">
                  <arg value="-verbose"/>

                  <!-- Set the classpath. -->
                  <arg value="-classpath"/>
                  <arg value="${class-dir}"/>

                  <!-- Generate a WSDL file. -->
                  <arg value="-wsdl"/>

                  <!-- Specify where to write other generated files. -->
                  <arg value="-d"/>
                  <arg value="${wsgen-outdir}"/>

                  <!-- Specify where to write WSDL and XML schema files. -->
                  <arg value="-r"/>
                  <arg value="${wsdl-outdir}"/>

                  <!-- Specify where to write generated source files. -->
                  <arg value="-s"/>
                  <arg value="${src-outdir}"/>

                  <!-- Keep generated source files. -->
                  <arg value="-keep"/>

                  <!-- Specify service endpoint interface/class. -->
                  <arg value="com.ivan.Calculator" />
            </exec>

            <!-- Delete the directory containing the generated classes. -->
            <delete dir="${gen-classdir}"/>
      </target>
</project>
```

When running the above Ant script, the following files will be produced:

- Add.java
  JAXB bean holding request data.
- AddResponse.java
  JAXB bean holding response data.
- CalculatorService_schema1.xsd
  XML schema defining elements wrapping request and response data in their XML form.
- CalculatorService.wsdl
  The WSDL document of the Calculator service.

When deploying a web service with a WSDL to GlassFish, the address of the web service will automatically be entered into the WSDL by the container. Other server containers may require specifying the address of the Calculator web service in the WSDL using the <soap:address> element, which would look like this:

```
...
    <service name="CalculatorService">
        <port name="CalculatorPort" binding="tns:CalculatorPortBinding">
            <soap:address
                location="http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService"/>
        </port>
    </service>
...
```

In order to be able to determine whether or not the JAXB bean classes are used or not, you can add some *println* statements and a default constructor, like in this example:

```
package com.ivan.javax;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement(name = "addNumbers", namespace = "http://www.ivan.com/calculator")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "addNumbers", namespace = "http://www.ivan.com/calculator", propOrder =
{
    "arg0", "arg1"
})
public class Add
{

    @XmlElement(name = "arg0", namespace = "")
    private int arg0;
    @XmlElement(name = "arg1", namespace = "")
    private int arg1;


    public Add()
    {
        super();
        System.out.println("***** In Add.constructor()");
    }

    public int getArg0()
    {
        System.out.println("***** In Add.getArg0()");
        return this.arg0;
    }

    public void setArg0(int arg0)
    {
        System.out.println("***** In Add.setArg0()");
        this.arg0 = arg0;
    }

    public int getArg1()
```

```
    {
        System.out.println("***** In Add.getArg1()");
        return this.arg1;
    }

    public void setArg1(int arg1)
    {
        System.out.println("***** In Add.setArg1()");
        this.arg1 = arg1;
    }
}
```

Deploying and Running the Service

We are now ready to deploy and run the calculator web service. No changes are required to the default web application deployment descriptor files (web.xml and sun-web.xml).
In Eclipse, the project can be deployed using Run As → Run on Server. GlassFish will automatically recognize and deploy the web service.
After having been deployed, the calculator service can be tested using the GlassFish web service testing facility. This testing facility will also enable us to see the SOAP request and response messages. Here is the request message:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header />
    <S:Body>
        <ns2:addNumbers xmlns:ns2="http://www.ivan.com/calculator">
            <arg0>2</arg0>
            <arg1>4</arg1>
        </ns2:addNumbers>
    </S:Body>
</S:Envelope>
```

...and here is the response to the above request:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:addNumbersResponse xmlns:ns2="http://www.ivan.com/calculator">
            <return>6</return>
        </ns2:addNumbersResponse>
    </S:Body>
</S:Envelope>
```

Compare the above messages with the generated XML schema:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
    version="1.0"
    targetNamespace="http://www.ivan.com/calculator"
    xmlns:tns="http://www.ivan.com/calculator"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="addNumbers" type="tns:addNumbers"/>

  <xs:element name="addNumbersResponse" type="tns:addNumbersResponse"/>

  <xs:complexType name="addNumbers">
    <xs:sequence>
      <xs:element name="arg0" type="xs:int"/>
      <xs:element name="arg1" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="addNumbersResponse">
    <xs:sequence>
      <xs:element name="return" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

If you added *println* statements and default constructors in the JAXB bean classes, as suggested above, you will be able to see that those classes are indeed instantiated but the getter and setter methods are never called. JAXB uses reflection to set the values in the beans.
If the @RequestWrapper and @ResponseWrapper annotations are removed from the service implementation class then the JAXB bean classes will not be used, as also can be seen in the console output.
Sample console output from the GlassFish application server running the web service. With println statemens in the JAXB bean classes and the @RequestWrapper and @ResponseWrapper annotations present in the service implementation class (parts of the log has been removed for clarity).

```
[#|2009-01-23T19:51:11.818|INFO| ***** In Add.constructor()|#]
[#|2009-01-23T19:51:11.849|INFO| Adding 5 and 7|#]
[#|2009-01-23T19:51:11.849|INFO| ***** In AddResponse.constructor()|#]
```

The first and third messages are printed from the JAXB bean classes and the second message is printed from the web service implementation class.

String Processor Web Service Example (Provider)

References:

JAX-WS 2.1 Specification, section 5.1

A web service can also be written to process entire protocol messages, usually SOAP, or message payloads, which in the case with SOAP is the contents of the SOAP body element.
Such a web service class will implement the *javax.xml.ws.Provider<T>* interface and be annotated with the @WebServiceProvider annotation.
In this example, we will develop such a web service that processes strings.
This example will make use of the XJC Plugin, which is available for Eclipse and IntelliJ IDEs. This plugin invokes the JAXB XML schema compiler to produce JAXB beans from an XML schema. Please download and install this plugin, if you haven't already.

Project Setup

The following steps describe how to set the String Processor project up in Eclipse:

- Create a Dynamic Web project.
- In the WEB-INF directory, create a directory named wsdl.
- In the wsdl directory, create a file named StringProcessorService_payloads.xsd
- In the wsdl directory, create a file named StringProcessorService.wsdl
- In the source hierarchy, create a package named *com.ivan.beans*.
- In the *com.ivan* package, create a Java class named *StringProcessor*.

Payloads XML Schema

The file StringProcessorService_payloads.xsd contains the XML schema that describes the payloads of the request and response messages received and returned by the service.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
    version="1.0"
    targetNamespace="http://www.ivan.com/stringprocessor"
    xmlns:tns="http://www.ivan.com/stringprocessor"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="reverseStringReq" type="tns:reverseStringReq"/>

  <xs:element name="reverseStringResp" type="tns:reverseStringResp"/>

  <xs:complexType name="reverseStringReq">
    <xs:sequence>
      <xs:element name="inString" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="reverseStringResp">
    <xs:sequence>
      <xs:element name="return" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

WSDL Document

The StringProcessorService.wsdl document is the document describing the different aspects of the String Processor web service. Note that the location of the service, in the <wsdl:service> element, needs to be modified according to personal setup.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
    targetNamespace="http://www.ivan.com/stringprocessor"
    name="StringProcessorService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.ivan.com/stringprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">

    <types>
        <xsd:schema>
            <xsd:import namespace="http://www.ivan.com/stringprocessor"
                schemaLocation="StringProcessorService_payloads.xsd" />
        </xsd:schema>
    </types>

    <message name="reverseStringRequest">
        <part name="parameters" element="tns:reverseStringReq" />
    </message>
    <message name="reverseStringResponse">
        <part name="result" element="tns:reverseStringResp" />
    </message>

    <portType name="StringProcessorPort">
        <operation name="reverseString">
            <input message="tns:reverseStringRequest" />
            <output message="tns:reverseStringResponse" />
        </operation>
    </portType>

    <binding name="StringProcessorPortBinding" type="tns:StringProcessorPort">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="reverseString">
            <soap:operation soapAction="urn:ReverseString" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>

    <service name="StringProcessorService">
        <port name="StringProcessorServicePort"
              binding="tns:StringProcessorPortBinding">
            <soap:address
                location="http://localhost:8080/JAX-
WS_WebServiceProvider/StringProcessorService" />
        </port>
    </service>
</definitions>
```

JAXB Beans

The XJC generator will, using the XML schema created earlier, generate the JAXB bean classes needed in this example project. I will assume that the XJC Eclipse plugin is to be used.

- Right-click the file StringProcessorService_payloads.xsd and select JAXB 2.1 → Run XJC.
- In the XJC Mandatory Parameters dialog, enter the package name com.ivan.beans and select the root source directory of the project as the output directory.
- Click the Finish button.
- Refresh the project in Eclipse.

The com.ivan.beans package should now contain the following four files:

- ObjectFactory.java
- package-info.java
- ReverseStringReq.java
- ReverseStringResp.java

In order for JAXB to be able to marshal the JAXB response bean to XML, we need to annotate the *ReverseStringResp* class with a @XmlRootElement annotation:

```
...
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "reverseStringResp", propOrder = {
    "_return"
})
@XmlRootElement
public class ReverseStringResp {
...
```

Service Provider Class

The service provider class is significantly different from the class implementing the web service we saw in the Calculator example above. First, we will have a look at the code and then some explanations will follow:

```
package com.ivan;

/**
 * This class implements a web service that processes strings.
 * The endpoint is implemented as a provider that receives and processes
 * message payloads (SOAP messages in this case), as opposed to web service
 * endpoint using the @WebService interface, which provides a service
 * endpoint interface and which receives already unmarshalled parameters.
 */
@WebServiceProvider(
    wsdlLocation="StringProcessorService.wsdl",
    portName="StringProcessorServicePort",
    serviceName="StringProcessorService",
    targetNamespace="http://www.ivan.com/stringprocessor")
@ServiceMode(value = Service.Mode.PAYLOAD)
@SOAPBinding(
    parameterStyle = SOAPBinding.ParameterStyle.WRAPPED,
    style = SOAPBinding.Style.DOCUMENT,
    use = SOAPBinding.Use.LITERAL)
public class StringProcessor implements Provider<Source>
{
    /* (non-Javadoc)
     * @see javax.xml.ws.Provider#invoke(java.lang.Object)
     */
    public Source invoke(final Source inRequestMessage)
    {
        Source theResponseSource = null;

        try
        {
```

```
            /* Prepare JAXB for marshaling and unmarshaling. */
            JAXBContext theJaxbContext =
                JAXBContext.newInstance("com.ivan.beans");
            Unmarshaller theUnmarshaller = theJaxbContext.createUnmarshaller();
            Marshaller theMarshaller = theJaxbContext.createMarshaller();

            /* Use JAXB to unmarshal the message payload into a request-bean. */
            JAXBElement<ReverseStringReq> theReqElem =
                (JAXBElement<ReverseStringReq>)theUnmarshaller
                    .unmarshal(inRequestMessage);
            ReverseStringReq theReq = theReqElem.getValue();

            System.out.println("Got a request to reverse the string: " +
                theReq.getInString());

            /*
             * Do the processing supplied by the operation - that is
             * reverse the string.
             */
            String theResultString =
                (new StringBuffer(theReq.getInString())).reverse().toString();

            System.out.println("Processed the result: " + theResultString);

            /* Create the response message payload and set the result. */
            ObjectFactory theObjFactory = new ObjectFactory();
            ReverseStringResp theResponse =
                theObjFactory.createReverseStringResp();
            theResponse.setReturn(theResultString);

            /*
             * Create a JAXB source that will marshal the supplied
             * JAXB bean to XML when the response message is assembled.
             */
            theResponseSource = new JAXBSource(theJaxbContext, theResponse);
        } catch (JAXBException theException)
        {
            theException.printStackTrace();
        }

        return theResponseSource;
    }
}
```

The @WebServiceProvider annotation is used when the service endpoint class implements the *Provider<T>* interface. As can be seen, the above class does not contain any methods that are exposed as operations of the web service, but just one single *invoke(Source)* method.
The *invoke* method will be invoked every time the web service receives a request and the protocol message or the message payload will be sent to the method using its only parameter.

The @ServiceMode annotation specifies whether the provider class will receive and produce entire protocol messages or just the message payloads. In this example, we will process and produce message payloads to make things more simple.

The @SOAPBinding annotation specifies the following things about the web service:

- Messaging style:
  style=SOAPBinding.Style.DOCUMENT
- Encoding:
  use=SOAPBinding.Use.LITERAL
- Parameter style:
  parameterStyle=SOAPBinding.ParameterStyle.WRAPPED

The above values are all default values of the SOAPBinding annotation and the annotation could thus have been omitted.

The provider class declaration:

```
public class StringProcessor implements Provider<Source>
```

Service endpoint classes may implement the *javax.xml.ws.Provider<T>* interface, allowing them to receive and process protocol messages or message payloads. as before.
The JAX-WS specification mandates support for the following providers:

- Provider<Source>
  In message payload mode. The *Source* interface is implemented by the following classes: *DOMSource*, *JAXBSource*, *SAXSource*, *StAXSource*, *StreamSource*
- Provider<SOAPMessage>
  In message mode.
- Provider<javax.activation.DataSource>

The following additional constrains are placed by the JAX-WS 2.1 specification on a provider class:

- Provider classes must implement a default constructor or no constructor.
- A provider class must be bound to a specific type.
  That is, it may implement Provider<Source>, Provider<SOAPMessage but implementing Provider<T> is not allowed.
- A provider class must be annotated with the @WebServiceProvider annotation.

Deploying and Running the Service

We are now ready to deploy and run the calculator web service. No changes are required to the default web application deployment descriptor files (web.xml and sun-web.xml).
In Eclipse, the project can be deployed using Run As → Run on Server. GlassFish will automatically recognize and deploy the web service.
GlassFish will not allow testing the web service, but it can be tested from within Eclipse by right-clicking the WSDL file and selecting Web Services → Test with Web Services Explorer.

4.8 JAX-WS Client Communications Models

Describe JAX-WS Client Communications Models.

References:
JAX-WS 2.1 Specification, section 4.3.2, 4.3.3, 4.3.5

In this section we will look at different ways for a client to invoke the calculator service developed above.

A JAX-WS client can use the following invocation models:

- Synchronous request-response
- Asynchronous request-response
- One-way

In order to be able to choose the mode in which to invoke a web service, an object implementing the *javax.xml.ws.Dispatch<T>* interface must be obtained from an instance of the *javax.xml.ws.Service*, which, among other things, acts as a factory for *Dispatch* instances. The *Dispatch* interface contains the following methods:

Method Name	Description
T invoke(T message)	Invoke a service operation synchronously sending the supplied message, or message payload. Returns the response message, or message payload.
Response<T> invokeAsync(T message)	Invokes a service operation asynchronously sending the supplied message, or message payload. Returns an object that can be polled for the result of the invocaion.
Future<?> invokeAsync(T message, AsyncHandler<T> handler)	Invokes a service operation asynchronously sending the supplied message, or message payload. The supplied handler will be called when a response to the operation is available. Returns an object that can be used to check the status of the operation invocation.
void invokeOneWay(T message)	Invokes a service operation using the one-way interaction mode. The invocation is logically non-blocking, but will, when HTTP is used as the underlying protocol, block until a HTTP response is received.

When requesting a *Dispatch* object, the client can also choose whether to work with entire messages, SOAP messages when SOAP is used, or message payloads only, which when SOAP is used is the contents of the SOAP body.

Synchronous Request-Response

As with normal method calls, synchronous request-response invocation of a web service operation blocks until the remote operation is completed and a response is returned. Synchronous invocation of service operations does not require the use of a *Dispatch* object, but can also be performed using a proxy obtained from an instance of *javax.xml.ws.Service*.
The first example shows how to invoke an operation using a synchronous request-response call on a proxy object. The *CalculatorService* class, which was automatically generated, extends the *Service* class.

```
    int theFirstNumber = 5;
    int theSecondNumber = 7;

    mCalculatorService = new CalculatorService();
    Calculator thePort = mCalculatorService.getCalculatorPort();
    int theResult = thePort.addNumbers(theFirstNumber, theSecondNumber);

    System.out.println("Adding " + theFirstNumber
        + " and " + theSecondNumber + " produced the result: " + theResult);
```

For detailed instructions on how to create a static standalone JAX-WS web service client, from which the above code fragment was taken, please refer to the section Static Clients below!

The next example shows how synchronous request-response invocation is done using a *Dispatch* object. Invocation of service operation is done in the method *callSyncReqRespService*:

```
    private final static String WSDL_LOCATION =
        "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl";
    private final static String SERVICE_NAMESPACE =
        "http://www.ivan.com/calculator";
    private final static String SERVICE_NAME = "CalculatorService";
    private final static String SERVICE_PORT_NAME = "CalculatorPort";

    private void callSyncReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<AddNumbers> theReqElem = createRequestPayload();

            /* Invoke the service operation using the Dispatch object. */
            JAXBElement<AddNumbersResponse> theResponse =
                (JAXBElement<AddNumbersResponse>)theDispatch.invoke(theReqElem);

            /* Retrieve the result from the response message. */
            int theSum = theResponse.getValue().getReturn();
            System.out.println("The sum calculated by the web service: "
                + theSum);

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private JAXBElement<AddNumbers> createRequestPayload()
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and
         * set parameters of the service operation invocation.
         */
        AddNumbers theAddRequestBean = theJaxbObjFactory.createAddNumbers();
        theAddRequestBean.setArg0(4);
        theAddRequestBean.setArg1(5);

        /*
         * Retrieve the JAXB element of the request operation.
```

```
         * This element is responsible for producing the XML of the
         * message payload when the message later is to be transmitted.
         */
        JAXBElement<AddNumbers> theReqElem =
            theJaxbObjFactory.createAddNumbers(theAddRequestBean);
        return theReqElem;
    }

    private Dispatch<Object> prepareCalculatorDispatch()
        throws MalformedURLException, JAXBException
    {
        QName theName =
            new QName(SERVICE_NAMESPACE, SERVICE_PORT_NAME);
        URL theWsdlUrl = new URL(WSDL_LOCATION);
        QName theServiceName = new QName(SERVICE_NAMESPACE, SERVICE_NAME);
        Service theService = Service.create(theWsdlUrl, theServiceName);

        JAXBContext theJaxbContext =
            JAXBContext.newInstance("com.ivan.calculator");

        /*
         * Note the constant Service.Mode.PAYLOAD parameter in the call
         * to the createDispatch method. This is where the client decides whether
         * it wants to work with entire messages (Service.Mode.MESSAGE) or
         * message payloads only (Service.Mode.PAYLOAD).
         */
        Dispatch<Object> theDispatch =
            theService.createDispatch(theName, theJaxbContext,
                Service.Mode.PAYLOAD);
        return theDispatch;
    }
```

Note that when using a *Dispatch* object to invoke a service operation, we must create either the entire (SOAP) message or, as in the above example, the message payload. Both the message and the message payload are usually in XML format.

Asynchronous Request-Response

Asynchronous invocation of a web service can be done either dynamically, using a *Dispatch* object, or by generating a proxy for the web service which also supports asynchronous invocation.

Dynamic Asynchronous Invocation

As described above, there are two methods available on a *Dispatch* object that enables us to invoke request-response service operations asynchronously. In the first example, we'll use the *invokeAsync* method that takes a message/payload as parameter and returns a *Response* object, which, as in the example, can be polled to determine whether a response is available. Helper methods are identical as in the above example on synchronous request-response invocation.

```
    private void callAsyncReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();
            JAXBElement<AddNumbers> theReqElem = createRequestPayload();

            /* Invoke the service operation using the Dispatch object. */
            Response<Object> theAsyncResponse = theDispatch.invokeAsync(theReqElem);

            /* Wait until there is a response to the request. */
            while(!theAsyncResponse.isDone())
            {
                System.out.println("Request is not done yet...");
            }
            System.out.println("Request is DONE!");

            JAXBElement<AddNumbersResponse> theResponse =
                (JAXBElement<AddNumbersResponse>) theAsyncResponse.get();
            int theSum = theResponse.getValue().getReturn();

            System.out.println("The sum calculated by the web service: " + theSum);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }
```

In the second example, an *AsyncHandler* object, containing a callback method, is also supplied when calling the *invokeAsync* method on the *Dispatch* object. The callback method is invoked when a result of the request is available. Again, helper methods remain the same as above and are not shown.

```
    private void callAsyncHandlerReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<AddNumbers> theReqElem = createRequestPayload();
            AsyncHandler<Object> theHandler = new AsyncHandler<Object>()
            {
                public void handleResponse(Response<Object> inResponse)
                {
                    try
                    {
                        System.out.println("Response received!");

                        JAXBElement<AddNumbersResponse> theResponse =
                            (JAXBElement<AddNumbersResponse>)inResponse.get();
                        int theSum = theResponse.getValue().getReturn();

                        System.out
                            .println("The sum calculated by the web service: "
                                + theSum);
                    } catch (Exception theException)
```

```
                    {
                        theException.printStackTrace();
                    }
                }
            };

            /* Invoke the service operation using the Dispatch object. */
            Future<?> theRequestFuture =
                theDispatch.invokeAsync(theReqElem, theHandler);

            while (!theRequestFuture.isDone())
            {
                System.out.println("Request is not done yet...");
            }
            System.out.println("Request is DONE!");

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }
```

Asynchronous Invocation with Proxies

Alternatively, asynchronous invocation can also be enabled for proxies. JAX-WS 2.1 does not specify any annotations by which a method in a web service can be marked as being asynchronous, instead the *enableAsyncMapping* binding declaration must be used. Such a declaration is usually placed in a separate file that refers to the WSDL:

```
<bindings xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    wsdlLocation="http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl"
    xmlns="http://java.sun.com/xml/ns/jaxws">

    <bindings node="//wsdl:portType[@name='Calculator']">
        <bindings node="wsdl:operation[@name='addNumbers']">
            <enableAsyncMapping>true</enableAsyncMapping>
        </bindings>
    </bindings>
</bindings>
```

The above example marks the operation *addNumbers* in the *Calculator* port as being asynchronous. By adding the -b option followed by the location of the above binding file when invoking *wsimport*, the generated port interface for the service will contain not only the *addNumbers* method, which invokes the service in a synchronous manner, but also two *addNumbersAsync* methods that allows for asynchronous invocation of the *addNumbers* method. The generated interface looks like this (parts not relevant to this example have been removed).

```
package com.ivan.calculator;


@WebService(
    name = "Calculator",
    targetNamespace = "http://www.ivan.com/calculator")
@XmlSeeAlso({ ObjectFactory.class })
public interface Calculator
{
    @WebMethod(operationName = "addNumbers", action = "urn:Add")
    @RequestWrapper(
        localName = "addNumbers",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbers")
    @ResponseWrapper(
        localName = "addNumbersResponse",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbersResponse")
    public Response<AddNumbersResponse> addNumbersAsync(
        @WebParam(name = "arg0", targetNamespace = "") int arg0,
        @WebParam(name = "arg1", targetNamespace = "") int arg1);
```

```
    @WebMethod(operationName = "addNumbers", action = "urn:Add")
    @RequestWrapper(
        localName = "addNumbers",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbers")
    @ResponseWrapper(
        localName = "addNumbersResponse",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbersResponse")
    public Future<?> addNumbersAsync(
        @WebParam(name = "arg0", targetNamespace = "")
        int arg0,
        @WebParam(name = "arg1", targetNamespace = "")
        int arg1,
        @WebParam(name = "asyncHandler", targetNamespace = "")
        AsyncHandler<AddNumbersResponse> asyncHandler);

    @WebMethod(action = "urn:Add")
    @WebResult(targetNamespace = "")
    @RequestWrapper(
        localName = "addNumbers",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbers")
    @ResponseWrapper(
        localName = "addNumbersResponse",
        targetNamespace = "http://www.ivan.com/calculator",
        className = "com.ivan.calculator.AddNumbersResponse")
    public int addNumbers(
        @WebParam(name = "arg0", targetNamespace = "")
        int arg0,
        @WebParam(name = "arg1", targetNamespace = "")
        int arg1);
}
```

The concept for invoking the asynchronous methods are the same as for corresponding methods in the *Dispatch* object. The two methods in the listing below shows how the web service is invoked asynchronously and results are retrieved. Parts of the class not relevant to this example have been removed.

```
package com.ivan.client;

/* Note that the class implements the AsyncHandler interface! */
public class CalculatorClient implements AsyncHandler<AddNumbersResponse>
{
    ...

    /*
     * This method shows how to call the calculator web service asynchronously
     * and later poll for results.
     */
    private void asyncPollCallService()
    {
        int theFirstNumber;
        int theSecondNumber;
        Response<AddNumbersResponse> theResponses[] =
            new Response[ADDING_NUMBERS.length];

        /*
         * Need to create the service in the code, since dependency
         * injection does not work outside of a container.
         */
        mCalculatorService = new CalculatorService();

        System.out.println("Service object: " + mCalculatorService);
        Calculator thePort = mCalculatorService.getCalculatorPort();

        /* Ask the web service to add some numbers for us. */
        for (int i = 0; i < ADDING_NUMBERS.length; i++)
        {
            theFirstNumber = ADDING_NUMBERS[i][0];
            theSecondNumber = ADDING_NUMBERS[i][1];

            System.out
                .println("Calling addNumbers for (" + theFirstNumber + ", "
```

```
                + theSecondNumber + ") at " + System.currentTimeMillis());
            /*
             * Save the response object, which we will later poll to get
             * the result of the addition.
             */
            theResponses[i] =
                thePort.addNumbersAsync(theFirstNumber, theSecondNumber);
        }

        System.out.println("Finished all invocations, checking results:");

        /* Poll for the results of the additions. */
        boolean theDoneFlags[] = new boolean[theResponses.length];
        for (int i = 0; i < theDoneFlags.length; i++)
        {
            theDoneFlags[i] = false;
        }
        int theDoneCount = 0;
        while (theDoneCount < theResponses.length)
        {
            for (int i = 0; i < theResponses.length; i++)
            {
                if (theResponses[i].isDone() && theDoneFlags[i] == false)
                {
                    try
                    {
                        System.out.println("Response done: "
                            + theResponses[i].get().getReturn() + " at "
                            + System.currentTimeMillis());
                        theDoneFlags[i] = true;
                        theDoneCount++;
                    } catch (Exception theException)
                    {
                        theException.printStackTrace();
                    }
                }
            }
        }
    }

    /*
     * This method shows how to call the calculator web service asynchronously
     * with results delivered using a callback mechanism.
     */
    private void asyncCallbackCallService()
    {
        int theFirstNumber;
        int theSecondNumber;

        /*
         * Need to create the service in the code, since dependency
         * injection does not work outside of a container.
         */
        mCalculatorService = new CalculatorService();

        System.out.println("Service object: " + mCalculatorService);
        Calculator thePort = mCalculatorService.getCalculatorPort();

        /* Ask the web service to add some numbers for us. */
        for (int i = 0; i < ADDING_NUMBERS.length; i++)
        {
            theFirstNumber = ADDING_NUMBERS[i][0];
            theSecondNumber = ADDING_NUMBERS[i][1];

            /*
             * In this example we call the web service client proxy and
             * provide a callback handler that will be called when a
             * result is available.
             */
            System.out
                .println("Calling addNumbers for (" + theFirstNumber + ", "
                    + theSecondNumber + ") at " + System.currentTimeMillis());
            thePort.addNumbersAsync(theFirstNumber, theSecondNumber, this);
        }

        System.out.println("Finished all invocations, waiting for results:");
```

```
        /*
         * Need to give the web service some time to reply.
         * If not, the program terminates and results are not reported.
         */
        try
        {
            Thread.sleep(10000L);
        } catch (InterruptedException theException)
        {
        }
    }

    /*
     * This is the callback method that will be invoked when a response
     * is available as a result of an asynchronous invocation of the
     * calculator web service.
     */
    public void handleResponse(final Response<AddNumbersResponse> inResponse)
    {
        try
        {
            System.out.println("Response received: "
                + inResponse.get().getReturn() + " at "
                + System.currentTimeMillis());
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }
}
```

One-Way

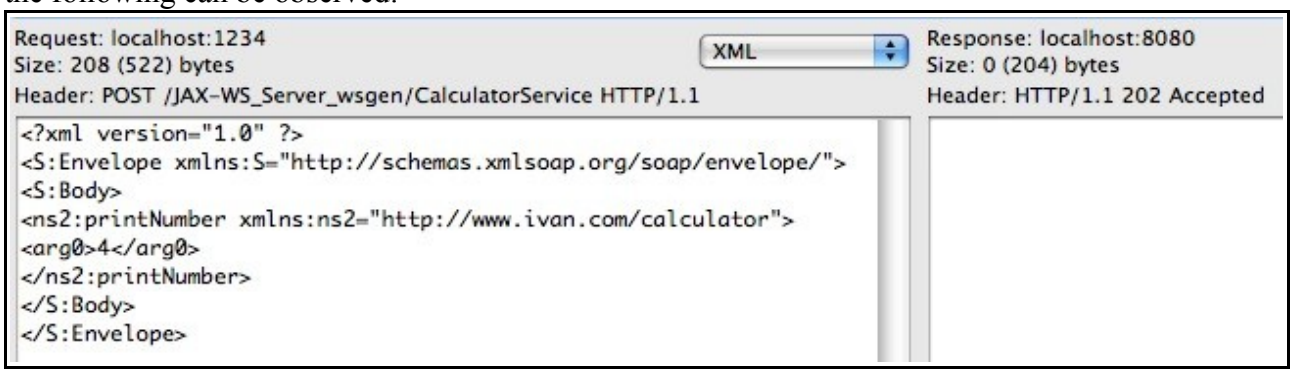
As a preparation, a new method is added to the Calculator web service developed previously and the web service is redeployed. The new version of the *Calculator* class now looks like this:

```
package com.ivan;


/**
 * Calculator service implementation class.
 */
@WebService(name = "Calculator", serviceName = "CalculatorService", targetNamespace =
"http://www.ivan.com/calculator", wsdlLocation = "CalculatorService.wsdl")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.WRAPPED, style =
SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.LITERAL)
public class Calculator
{
    /* Constant(s): */

    /* Instance variable(s): */
    @Resource
    private WebServiceContext mWSContext;

    /**
     * Initializes the web service.
     */
    @PostConstruct
    @WebMethod(exclude = true)
    public void init()
    {
        System.out.println("Web service initialized, got service context: "
            + mWSContext);
    }

    /**
     * Prints the supplied numbers without returning any result.
     *
     * @param inNumber Number to print.
     */
    @Oneway
    @RequestWrapper(className = "com.ivan.javax.PrintNumber", localName = "printNumber",
      targetNamespace = "http://www.ivan.com/calculator")
    @WebMethod(operationName = "printNumber", action = "urn:PrintNumber")
    public void printNumber(final int inNumber)
    {
        System.out.println("Printing number: " + inNumber);
    }

    /**
     * Adds the supplied numbers.
     *
     * @param inNumber1 First number to add.
     * @param inNumber2 Second number to add.
     * @return Sum of the two numbers.
     */
    @WebMethod(operationName = "addNumbers", action = "urn:Add")
    @ResponseWrapper(className = "com.ivan.javax.AddResponse", localName =
      "addNumbersResponse", targetNamespace = "http://www.ivan.com/calculator")
    @RequestWrapper(className = "com.ivan.javax.Add", localName = "addNumbers",
      targetNamespace = "http://www.ivan.com/calculator")
    public int add(final int inNumber1, final int inNumber2)
    {
        System.out.println("Adding " + inNumber1 + " and " + inNumber2);

        return inNumber1 + inNumber2;
    }
}
```

The JAXB wrapper classes in the client are generated from the new WSDL and the following two methods are added to the calculator client class:

```
    private void callOnewayService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<PrintNumber> theReqElem = createOnewayRequestPayload();

            /* Invoke the service operation using the Dispatch object. */
            theDispatch.invokeOneWay(theReqElem);

            System.out
                .println("One-way invocation of the service yields no response.");

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private JAXBElement<PrintNumber> createOnewayRequestPayload()
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and set
         * parameter of the service operation invocation.
         */
        PrintNumber thePrintReqBean = theJaxbObjFactory.createPrintNumber();
        thePrintReqBean.setArg0(4);

        /*
         * Retrieve the JAXB element of the request operation. This
         * element is responsible for producing the XML of the message
         * payload when the message later is to be transmitted.
         */
        JAXBElement<PrintNumber> theReqElem =
            theJaxbObjFactory.createPrintNumber(thePrintReqBean);
        return theReqElem;
    }
```

If the communication between the client and server is examined using the Eclipse TCP/IP monitor, the following can be observed:



The request is wrapped, as expected, and the response is empty. The HTTP/1.1 code returned after a successful invocation of the one-way operation is 202.

4.9 JAX-WS Web Service Clients

Given an set of requirements, design and develop a Web service client, such as a JavaEE client and a stand-alone client, using JAX-WS.

A JAX-WS web service client can be either a dynamic client, for which no artifacts have been generated, or a static client for which a set of artifacts have been generated. JAX-WS web service clients can further be grouped into standalone clients and JavaEE clients.

Dynamic Clients

Dynamic clients can choose to work either with protocol-specific message structures, e.g. SOAP messages for the SOAP protocol binding, or with message payloads, which in the case of SOAP would be the contents of the SOAP body in a message.
Here, the complete class containing all the examples from the above section on JAX-WS Client Communication Models is shown.

```
package com.ivan.dynamic;

/**
 * This class implements a JAX-WS dymanic web service client that does
 * not require any generated artifacts.
 */
public class DynamicCalculatorClient
{
    /* Constant(s): */
    private final static String WSDL_LOCATION =
        "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl";
    private final static String SERVICE_NAMESPACE =
        "http://www.ivan.com/calculator";
    private final static String SERVICE_NAME = "CalculatorService";
    private final static String SERVICE_PORT_NAME = "CalculatorPort";

    public static void main(String[] args)
    {
        DynamicCalculatorClient theClient = new DynamicCalculatorClient();
        theClient.callOnewayService();
    }

    private void callAsyncHandlerReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<AddNumbers> theReqElem = createRequestPayload();
            AsyncHandler<Object> theHandler = new AsyncHandler<Object>()
            {
                public void handleResponse(Response<Object> inResponse)
                {
                    try
                    {
                        System.out.println("Response received!");

                        JAXBElement<AddNumbersResponse> theResponse =
                            (JAXBElement<AddNumbersResponse>)inResponse.get();
                        int theSum = theResponse.getValue().getReturn();

                        System.out
                            .println("The sum calculated by the web service: "
                                + theSum);
                    } catch (Exception theException)
                    {
                        theException.printStackTrace();
                    }
                }
            };

            /* Invoke the service operation using the Dispatch object. */
```

```
            Future<?> theRequestFuture =
                theDispatch.invokeAsync(theReqElem, theHandler);

            while (!theRequestFuture.isDone())
            {
                System.out.println("Request is not done yet...");
            }
            System.out.println("Request is DONE!");

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private void callAsyncReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<AddNumbers> theReqElem = createRequestPayload();

            /* Invoke the service operation using the Dispatch object. */
            Response<Object> theAsyncResponse =
                theDispatch.invokeAsync(theReqElem);

            while (!theAsyncResponse.isDone())
            {
                System.out.println("Request is not done yet...");
            }
            System.out.println("Request is DONE!");

            JAXBElement<AddNumbersResponse> theResponse =
                (JAXBElement<AddNumbersResponse>)theAsyncResponse.get();
            int theSum = theResponse.getValue().getReturn();

            System.out.println("The sum calculated by the web service: "
                + theSum);

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private void callSyncReqRespService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<AddNumbers> theReqElem = createRequestPayload();

            /* Invoke the service operation using the Dispatch object. */
            JAXBElement<AddNumbersResponse> theResponse =
                (JAXBElement<AddNumbersResponse>)theDispatch.invoke(theReqElem);

            int theSum = theResponse.getValue().getReturn();

            System.out.println("The sum calculated by the web service: "
                + theSum);

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private void callOnewayService()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            JAXBElement<PrintNumber> theReqElem = createOnewayRequestPayload();
```

```
            /* Modify the endpoint address to go through Eclipse TCP/IP monitor. */
//            Map<String, Object> theReqContext = theDispatch.getRequestContext();
//            theReqContext.put(Dispatch.ENDPOINT_ADDRESS_PROPERTY,
//                "http://localhost:1234/JAX-WS_Server_wsgen/CalculatorService");

            /* Invoke the service operation using the Dispatch object. */
            theDispatch.invokeOneWay(theReqElem);

            System.out.println("One-way invocation of the service yields no response.");

        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private Dispatch<Object> prepareCalculatorDispatch()
        throws MalformedURLException, JAXBException
    {
        QName theName = new QName(SERVICE_NAMESPACE, SERVICE_PORT_NAME);
        URL theWsdlUrl = new URL(WSDL_LOCATION);
        QName theServiceName = new QName(SERVICE_NAMESPACE, SERVICE_NAME);
        Service theService = Service.create(theWsdlUrl, theServiceName);

        JAXBContext theJaxbContext = JAXBContext.newInstance("com.ivan.calculator");

        Dispatch<Object> theDispatch =
            theService.createDispatch(theName, theJaxbContext,
                Service.Mode.PAYLOAD);

        return theDispatch;
    }

    private JAXBElement<AddNumbers> createRequestPayload()
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and set
         * parameters of the service operation invocation.
         */
        AddNumbers theAddRequestBean = theJaxbObjFactory.createAddNumbers();
        theAddRequestBean.setArg0(4);
        theAddRequestBean.setArg1(5);

        /*
         * Retrieve the JAXB element of the request operation. This
         * element is responsible for producing the XML of the message
         * payload when the message later is to be transmitted.
         */
        JAXBElement<AddNumbers> theReqElem =
            theJaxbObjFactory.createAddNumbers(theAddRequestBean);
        return theReqElem;
    }

    private JAXBElement<PrintNumber> createOnewayRequestPayload()
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and set
         * parameter of the service operation invocation.
         */
        PrintNumber thePrintReqBean = theJaxbObjFactory.createPrintNumber();
        thePrintReqBean.setArg0(4);

        /*
         * Retrieve the JAXB element of the request operation. This
         * element is responsible for producing the XML of the message
         * payload when the message later is to be transmitted.
         */
        JAXBElement<PrintNumber> theReqElem =
            theJaxbObjFactory.createPrintNumber(thePrintReqBean);
        return theReqElem;
    }
}
```

Static Clients

In this section we will develop a standalone, static web service client to the Calculator web service developed above.
First all the static artifacts needed are generated using the wsimport tool. The following Ant script invokes wsimport for us. Note that the Calculator web service must be running when generating the artifacts, in order for wsimport to be able to access the WSDL document of the service.

```
<?xml version="1.0"?>

<project default="main" basedir=".">

      <property name="class-dir" value="${basedir}/bin" />
      <property name="wsimport-outdir" value="${basedir}/wsimport_generated" />
      <property name="gen-classdir" value="${wsimport-outdir}/com/" />
      <property name="src-outdir" value="${basedir}/src/" />
      <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport" />
      <property name="wsdl-location" value="http://localhost:8080/JAX-
WS_Server_wsgen/CalculatorService?wsdl" />

      <echo message="calling the client generation task wsimport" />
      <target name="main">
            <exec executable="${wsimport-cmd}">
                  <arg value="-verbose" />

                  <!-- Specify where to write generated class files. -->
                  <arg value="-d" />
                  <arg value="${wsimport-outdir}" />

                  <!-- Specify where to write generated source files. -->
                  <arg value="-s" />
                  <arg value="${src-outdir}" />

                  <!-- Keep generated source files. -->
                  <arg value="-keep" />

                  <!-- Specify location of WSDL from which to generate stuff. -->
                  <arg value="${wsdl-location}" />
            </exec>

            <!--
            Delete the directory containing the generated classes and its
            contents.
        -->
            <delete dir="${gen-classdir}" />
      </target>
</project>
```

Running the above Ant script will generate the following files in the client project, all located in the *com.ivan.calculator* package:

- AddNumbers.java
- AddNumbersResponse.java
- Calculator.java
- CalculatorService.java
- ObjectFactory.java
- package-info.java
- PrintNumber.java

With the generated files in place, we can now write the client:

```
package com.ivan.client;

import javax.xml.ws.WebServiceRef;

import com.ivan.calculator.Calculator;
import com.ivan.calculator.CalculatorService;

/**
 * Standalone static JAX-WS client invoking the Calculator service.
 */
public class CalculatorClient
{
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl")
    private CalculatorService mCalculatorService;


    public static void main(String[] args)
    {
        CalculatorClient theClient = new CalculatorClient();
        theClient.callService();
    }

    private void callService()
    {
        int theFirstNumber = 5;
        int theSecondNumber = 7;

        /*
         * Need to create the service in the code, since there seem to
         * be problems with the dependency injection.
         */
        mCalculatorService = new CalculatorService();

        System.out.println("Service object: " + mCalculatorService);
        Calculator thePort = mCalculatorService.getCalculatorPort();
        System.out.println("Invoking the addNumbers operation on the service.");
        int theResult = thePort.addNumbers(theFirstNumber, theSecondNumber);
        System.out.println("Adding " + theFirstNumber
            + " and " + theSecondNumber + " produced the result: " + theResult);
    }
}
```

Note the annotation on the instance variable *mCalculatorService*:

```
...
@WebServiceRef(wsdlLocation =
    "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl")
...
```

With the proper measures taken, this annotation is supposed to be able to create and inject a reference to a *CalculatorService* instance. As of writing this, I have not been successful in using this feature, so I have reverted to explicitly creating an instance of the *CalculatorService* class in the code.

Standalone Clients

See the above section on Dynamic Clients and the section on Static Clients!

JavaEE Clients

Accessing a JAX-WS web service from within an application server is very similar to doing it from a standalone client. The main difference, in my case, is that the dependency injection works without glitches.
Below follows an example of how to access the Calculator web service from a servlet and a JSP. Both were developed in the same project.

- Create a Dynamic Web project in Eclipse using the GlassFish application server.
  In NetBeans, create a Web Application project.
- Copy the generated artifacts from the web service client described in the section Static Clients above to the dynamic web project.
- Implement a servlet as follows:

```
package com.ivan.servlet;

/**
 * This class implements a servlet that accesses a JAX-WS web service.
 */
public class JAXWSClientServlet extends HttpServlet
{
    /* Constant(s): */
    private static final long serialVersionUID = 1L;

    /* Instance variable(s): */
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl")
    private CalculatorService mCalculatorService;


    private int calculateSum(final int inNumber1, final int inNumber2)
    {
        Calculator thePort = mCalculatorService.getCalculatorPort();
        int theResult = thePort.addNumbers(inNumber1, inNumber2);

        return theResult;
    }

    protected void doGet(HttpServletRequest inRequest,
        HttpServletResponse inResponse) throws ServletException, IOException
    {
        int theFirstNumber = 5;
        int theSecondNumber = 7;

        /* If there are parameters from the query string, use those. */
        String theNumStr1 = inRequest.getParameter("theFirstNumber");
        String theNumStr2 = inRequest.getParameter("theSecondNumber");
        if ((theNumStr1 != null) && (theNumStr2 != null))
        {
            theFirstNumber = Integer.parseInt(theNumStr1);
            theSecondNumber = Integer.parseInt(theNumStr2);
        }

        /* Calculate the sum of the numbers using the web service. */
        int theSum = calculateSum(theFirstNumber, theSecondNumber);

        /* Output the result. */
        inResponse.setContentType("text/html;charset=UTF-8");
        PrintWriter out = inResponse.getWriter();
        out.println("<html><head>");
        out.println("<title>JAX-WS Servlet Client</title></head>");
        out.println("<body>");
        out.println("Adding the numbers " + theFirstNumber + " and " +
            theSecondNumber + " produced the result: "  + theSum);

        out.println("</body>");
        out.println("</html>");
        out.close();

    }
}
```

- Modify, alternatively create, the web.xml deployment descriptor file.
  The result should be:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
    <display-name>JAX-WS_JavaEE_Client</display-name>

    <servlet>
        <description>JAX-WS Web Service Client Servlet</description>
        <display-name>JAXWSClientServlet</display-name>
        <servlet-name>JAXWSClientServlet</servlet-name>
        <servlet-class>com.ivan.servlet.JAXWSClientServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>JAXWSClientServlet</servlet-name>
        <url-pattern>/servlet.do</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

- Create the JSP file index.jsp.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ page import="com.ivan.calculator.Calculator" %>
<%@ page import="com.ivan.calculator.CalculatorService" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%!
public void jspInit()
{
    /*
     * If the application context does not contain a calculator service
     * port object, then get one and save it in the application context.
     */
    ServletContext theContext = getServletConfig().getServletContext();

    if (theContext.getAttribute("calculatorPort") == null)
    {
        CalculatorService mCalculatorService = new CalculatorService();
        Calculator thePort = mCalculatorService.getCalculatorPort();
        theContext.setAttribute("calculatorPort", thePort);
    }
}
%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>JAX-WS JSP Web Service Client</title>
</head>
<body>
    <p>
    <c:out value="Welcome to the JSP page accessing the Calculator web service!"/>
    </p>
    <c:out value="To add custom numbers, append them as parameters to the request
string."/>
    <br/>
    <c:out value="For instance, to add the numbers 1 and 2, append:"/>
    <br/>
    <c:out value="?theFirstNumber=1&theSecondNumber=2"/>
    <p/>

    <br/><br/>
    <%
    int theFirstNumber = 5;
```

```
    int theSecondNumber = 7;

    /* If there are parameters from the query string, use those. */
    String theNumStr1 = request.getParameter("theFirstNumber");
    String theNumStr2 = request.getParameter("theSecondNumber");
    if ((theNumStr1 != null) && (theNumStr2 != null))
    {
        theFirstNumber = Integer.parseInt(theNumStr1);
        theSecondNumber = Integer.parseInt(theNumStr2);
    }

    /* Retrieve the calculator web service port from the application context. */
    Calculator thePort = (Calculator)application.getAttribute("calculatorPort");
    int theResult = thePort.addNumbers(theFirstNumber, theSecondNumber);

    /* Store the numbers and the result in the request scope. */
    request.setAttribute("theResult", new Integer(theResult));
    request.setAttribute("theFirstNumber", new Integer(theFirstNumber));
    request.setAttribute("theSecondNumber", new Integer(theSecondNumber));
    %>

    <c:out value="Adding the numbers ${theFirstNumber} and ${theSecondNumber} produced
the result ${theResult}."/>
</body>
</html>
```

- Deploy the web application.
- Access the JSP using the URL (may vary depending on the context path):
  http://localhost:8080/JAX-WS_JavaEE_Client/index.jsp?theFirstNumber=55&theSecondNumber=76
- Access the servlet using the URL (may vary depending on the context path):
  http://localhost:8080/JAX-WS_JavaEE_Client/servlet.do?theFirstNumber=44&theSecondNumber=76

4.10 Clients of Stateful Web Services

Given a set of requirements, create and configure a Web service client that accesses a stateful Web service.

References:
JAX-WS 2.1 Specification, section 1.3.9, 4.2.1.1, 4.2.3.1, 10.4.1.4, 11.3.3,
https://jax-ws.dev.java.net/jax-ws-21-ea3/docs/statefulWebservice.html

A service using a SOAP/HTTP binding can use the following three mechanisms for maintaining a session:

- Cookies
- URL Rewriting
- SSL Sessions

The WS-I Basic Profile says the following about stateful web services:

- BP allows a service to use HTTP cookies to maintain sessions.
- BP discourages a service to rely on the use of cookies for maintaining a state.

Example

In this example we will see how a session of a client to the Calculator web service is maintained across requests.
First of all, the web service is modified to output the current session id by adding the following method to the endpoint implementation class and invoking it from the *add(...)* and *printNumber(...)* methods:

```
    private void printHTTPSessionId()
    {
        Map<String, Object> theContext = mWSContext.getMessageContext();
        HttpServletRequest theReq =
            (HttpServletRequest)theContext.get(MessageContext.SERVLET_REQUEST);

        /* Get HTTP session only if there already exists one. */
        HttpSession theSession = theReq.getSession();
        String theSessionId = "[no HTTP session available]";
        if (theSession != null)
        {
            theSessionId = theSession.getId();
        }
        System.out.println("Session ID: " + theSessionId);
    }
```

Example of use in the *printNumber(...)* method in the *Calculator* endpoint implementation class:

```
    /**
     * Prints the supplied numbers without returning any result.
     *
     * @param inNumber Number to print.
     */
    @RequestWrapper(className = "com.ivan.javax.PrintNumber", localName = "printNumber",
        targetNamespace = "http://www.ivan.com/calculator")
    @WebMethod(operationName = "printNumber", action = "urn:PrintNumber")
    public void printNumber(final int inNumber)
    {
        printHTTPSessionId();

        System.out.println("***** Printing number: " + inNumber);
    }
```

Important! The @Oneway annotation on the *printNumber* method must be removed or commented out!
If this is not done, the HTTP response will be closed before entering the *printNumber* method and thus a HTTP session cannot be created (an exception will occur).
In the case that an exception occurs despite having removed the @Oneway annotation, try declaring "throws Exception" on the *printNumber* method, to ensure that it becomes a request-response type operation.

To configure a client of a web service, set the *javax.xml.ws.session.maintain* property to true. The first example shows how to do it in a static web service client:

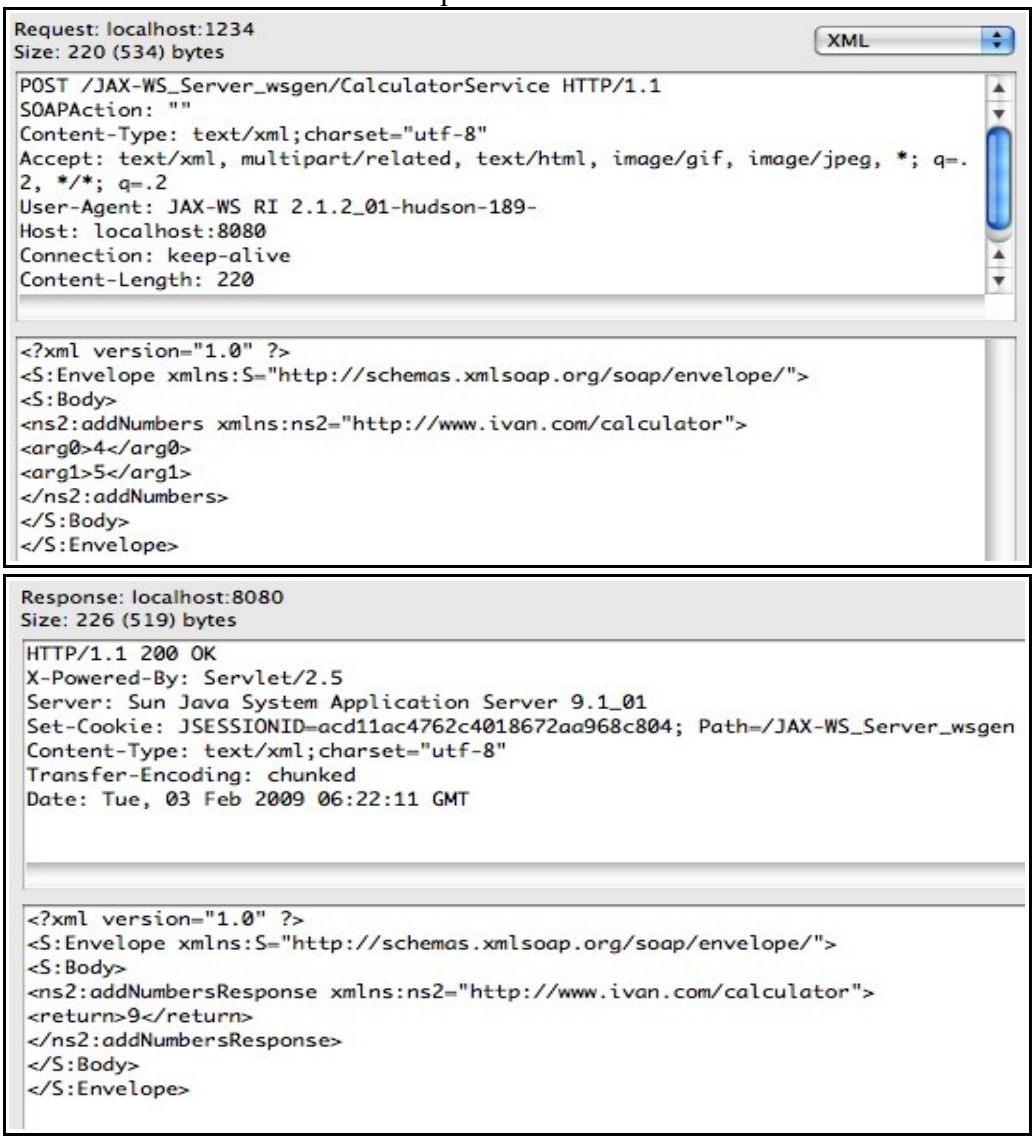
```
    private final static int[][] ADDING_NUMBERS = {{1, 3}, {5, 7}, {10, 11}};

    private void callService()
    {
        int theFirstNumber;
        int theSecondNumber;

        /* Need to create the service in the code, since there seem to
         * be problems with the dependency injection.
         */
        mCalculatorService = new CalculatorService();
        Calculator thePort = mCalculatorService.getCalculatorPort();

        /*
         * Enable maintaining of a session when interacting with the service.
         * Note that the property only have to be set once in the request
         * context and then applies to all subsequent requests.
         * This is the javax.xml.ws.session.maintain property.
         */
        ((BindingProvider)thePort).getRequestContext().put(
            BindingProvider.SESSION_MAINTAIN_PROPERTY, true);

        /* Add several numbers to see if the session remains the same. */
        for (int i = 0; i < ADDING_NUMBERS.length; i++)
        {
            theFirstNumber = ADDING_NUMBERS[i][0];
            theSecondNumber = ADDING_NUMBERS[i][1];

            int theResult = thePort.addNumbers(theFirstNumber, theSecondNumber);
            System.out.println("Adding " + theFirstNumber + " and "
                + theSecondNumber + " produced the result: " + theResult);

            /* Invoke another service operation to see if is the same session. */
            thePort.printNumber(theResult);
        }
    }
```

With the modifications to the Calculator service and the static client in place, the following console output can be seen on the server side when the client is run (parts of the log have been omitted for clarity):

```
[#|2009-02-03T20:41:08.373|INFO| Adding 1 and 3|#]
[#|2009-02-03T20:41:08.374|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.415|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.416|INFO| ***** Printing number: 4|#]
[#|2009-02-03T20:41:08.418|INFO| Adding 5 and 7|#]
[#|2009-02-03T20:41:08.418|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.422|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.422+0800|INFO| ***** Printing number: 12|#]
[#|2009-02-03T20:41:08.424+0800|INFO| Adding 10 and 11|#]
[#|2009-02-03T20:41:08.424+0800|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.429+0800|INFO| Session ID: a399d96431bd504d3088e94cb7f2|#]
[#|2009-02-03T20:41:08.429+0800|INFO| ***** Printing number: 21|#]
```

We see that all the requests to the two different web service operations are both within the same session.

If we use the Eclipse TCP/IP monitor to look at the data exchange of the first request-response, we can see that a cookie is returned with the response:



The second example shows how a dynamic client accesses a stateful web service:

```
package com.ivan.dynamic;

/**
 * This class implements a JAX-WS dymanic web service client that does
 * not require any generated artifacts.
 */
public class DynamicCalculatorClient
{
    /* Constant(s): */
    private final static String WSDL_LOCATION =
        "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl";
    private final static String SERVICE_NAMESPACE =
        "http://www.ivan.com/calculator";
    private final static String SERVICE_NAME = "CalculatorService";
    private final static String SERVICE_PORT_NAME = "CalculatorPort";
    private final static int FIRST_NUMBER_5 = 5;
    private final static int SECOND_NUMBER_4 = 4;
    private final static int[][] ADDING_NUMBERS = {{1, 3}, {5, 7}, {10, 11}};

    /* Instance variable(s): */

    /**
     * @param args
     */
    public static void main(String[] args)
    {
        DynamicCalculatorClient theClient = new DynamicCalculatorClient();
        theClient.callSessionServiceOperations();
    }

    private void callSessionServiceOperations()
    {
        try
        {
            Dispatch<Object> theDispatch = prepareCalculatorDispatch();

            /*
             * Set the property indicating that client wants to participate
             * in a session with the service endpoint.
             */
            Map<String, Object> theReqContext = theDispatch.getRequestContext();
            theReqContext.put(BindingProvider.SESSION_MAINTAIN_PROPERTY,
                Boolean.TRUE);

            /* Modify the endpoint address to go through TCP/IP monitor. */
            theReqContext.put(Dispatch.ENDPOINT_ADDRESS_PROPERTY,
                "http://localhost:1234/JAX-WS_Server_wsgen/CalculatorService");

            /* Add several numbers to see if the session remains the same. */
            for (int i = 0; i < ADDING_NUMBERS.length; i++)
            {
                int theFirstNumber = ADDING_NUMBERS[i][0];
                int theSecondNumber = ADDING_NUMBERS[i][1];

                JAXBElement<AddNumbers> theReqElem =
                    createRequestPayload(theFirstNumber, theSecondNumber);

                /* Invoke the service operation using the Dispatch object. */
                JAXBElement<AddNumbersResponse> theResponse =
                    (JAXBElement<AddNumbersResponse>)theDispatch
                        .invoke(theReqElem);

                int theResult = theResponse.getValue().getReturn();

                System.out.println("Adding " + theFirstNumber + " and "
                    + theSecondNumber + " produced the result: " + theResult);

                /* Invoke another operation on the service. */
                JAXBElement<PrintNumber> theOneWayReqElem =
                    createOnewayRequestPayload(theResult);
                theDispatch.invokeOneWay(theOneWayReqElem);
            }
        } catch (Exception theException)
        {
            theException.printStackTrace();
```

```
        }
    }

    private Dispatch<Object> prepareCalculatorDispatch()
        throws MalformedURLException, JAXBException
    {
        QName theName = new QName(SERVICE_NAMESPACE, SERVICE_PORT_NAME);
        URL theWsdlUrl = new URL(WSDL_LOCATION);
        QName theServiceName = new QName(SERVICE_NAMESPACE, SERVICE_NAME);
        Service theService = Service.create(theWsdlUrl, theServiceName);

        JAXBContext theJaxbContext =
            JAXBContext.newInstance("com.ivan.calculator");

        Dispatch<Object> theDispatch =
            theService.createDispatch(theName, theJaxbContext,
                Service.Mode.PAYLOAD);

        return theDispatch;
    }

    private JAXBElement<AddNumbers> createRequestPayload(
        final int inFirstNumber, final int inSecondNumber)
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and set
         * parameters of the service operation invocation.
         */
        AddNumbers theAddRequestBean = theJaxbObjFactory.createAddNumbers();
        theAddRequestBean.setArg0(inFirstNumber);
        theAddRequestBean.setArg1(inSecondNumber);

        /*
         * Retrieve the JAXB element of the request operation. This
         * element is responsible for producing the XML of the message
         * payload when the message later is to be transmitted.
         */
        JAXBElement<AddNumbers> theReqElem =
            theJaxbObjFactory.createAddNumbers(theAddRequestBean);
        return theReqElem;
    }

    private JAXBElement<PrintNumber> createOnewayRequestPayload(
        final int inNumber)
    {
        ObjectFactory theJaxbObjFactory = new ObjectFactory();

        /*
         * Create the JAXB bean holding the request payload and set
         * parameter of the service operation invocation.
         */
        PrintNumber thePrintReqBean = theJaxbObjFactory.createPrintNumber();
        thePrintReqBean.setArg0(inNumber);

        /*
         * Retrieve the JAXB element of the request operation. This
         * element is responsible for producing the XML of the message
         * payload when the message later is to be transmitted.
         */
        JAXBElement<PrintNumber> theReqElem =
            theJaxbObjFactory.createPrintNumber(thePrintReqBean);
        return theReqElem;
    }
}
```

Looking at the HTTP headers of all the requests and responses using the TCP/IP monitor, we can see that a cookies is set in the first response and the same cookie is enclosed in all the subsequent requests to the web service:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_01
Set-Cookie: JSESSIONID=ecd9389fe12cd017f3fe2e63ae00; Path=/JAX-WS_Server_wsgen
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Wed, 04 Feb 2009 20:01:14 GMT
```

HTTP header of the response of the first request to the web service.

```
POST /JAX-WS_Server_wsgen/CalculatorService HTTP/1.1
SOAPAction: ""
Content-Type: text/xml;charset="utf-8"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2, */*;
q=.2
User-Agent: JAX-WS RI 2.1.2_01-hudson-189-
Cookie: JSESSIONID=ecd9389fe12cd017f3fe2e63ae00
Host: localhost:8080
Connection: keep-alive
Content-Length: 208
```

HTTP header of the second request to the web service.

```
POST /JAX-WS_Server_wsgen/CalculatorService HTTP/1.1
SOAPAction: ""
Content-Type: text/xml;charset="utf-8"
Accept: text/xml, multipart/related, text/html, image/gif, image/jpeg, *; q=.2, */*;
q=.2
User-Agent: JAX-WS RI 2.1.2_01-hudson-189-
Cookie: JSESSIONID=ecd9389fe12cd017f3fe2e63ae00
Host: localhost:8080
Connection: keep-alive
Content-Length: 220
```

HTTP header of the third request to the web service.

Subsequent requests have been omitted for brevity.

Looking at the console output from the web service server, we see that the output is almost identical, apart from time and session id of course, to that from the static client example above. Parts of the output have been omitted for brevity and clarity.

```
[#|2009-02-04T20:16:17.422+0800|INFO| Adding 1 and 3|#]
[#|2009-02-04T20:16:17.423+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.474+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.475+0800|INFO| ***** Printing number: 4|#]
[#|2009-02-04T20:16:17.479+0800|INFO| Adding 5 and 7|#]
[#|2009-02-04T20:16:17.479+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.485+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.485+0800|INFO| ***** Printing number: 12|#]
[#|2009-02-04T20:16:17.489+0800|INFO| Adding 10 and 11|#]
[#|2009-02-04T20:16:17.489+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.496+0800|INFO| Session ID: edb5c8ff0690803c51cbc555d13f|#]
[#|2009-02-04T20:16:17.496+0800|INFO| ***** Printing number: 21|#]
```

5. REST, JSON, SOAP and XML Processing APIs (JAXP, JAXB and SAAJ)

5.1 REST Web Services

Describe the characteristics of REST Web Services.

The central concept of REST web services is resources. Resources can be things in a system that represent some real-life concept, for instance, a customer, a book, an author etc. A REST web service provides a way to retrieve a representation of such a resource and to manipulate resources using a limited set of verbs.
A request to a REST web service usually consists of a HTTP request. The parts of the request of main interest are the URI and the HTTP method. For instance, if a HTTP GET request is issued to the URI http://www.ivan.com/bookservice/authors/1 this may be a request to retrieve (GET) the author with the id 1 from the book service at www.ivan.com.
The following are some characteristics of REST web services:

- Addressability
  The information that selects which resource to operate on is kept in an URI.
  A resource is uniquely identified by a URI, but a resource may have several different URIs. For instance, the author in the above example may also be accessible through a book which he/she has written:
  http://www.ivan.com/bookservice/books/SomeBookTitle/author
- Uniform Interface
  This means that there are a limited set of operations available. When used in conjunction with HTTP, these operations are the HTTP operations GET, POST, PUT, DELETE etc.
- Statelessness
  Every request to a REST web service happens in complete isolation. The server does not maintain any state associated to a client between requests.
- Connectedness
  Resources are connected to each other using URIs and this enables navigation among the resources, without prior knowledge of their structure. Compare this to navigating web pages by following links on the different pages.
  For instance, when requesting information about a book from the book service, the representation of the book contains one or more URIs linking to the author(s) of the book. This also allows for passing references to resources to other parts of a system etc.
- Representations
  Resources are manipulated by exchanging representations of the resources.
  For instance, an XML representation is created from a book object and returned to a client requesting information on that particular book.

5.2 JSON Web Services

Describe the characteristics of JSON Web Services.

References:
http://www.json.org/
https://jax-ws-commons.dev.java.net/json/
http://weblogs.java.net/blog/kohsuke/archive/2007/04/jaxws_ri_extens.html

JSON Encoding Format

JSON, JavaScript Object Notation, is a lightweight text-based data interchange format. The following shows an example of a JSON and a XML representation of the same data:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}


<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```

Libraries that implement reading and writing of JSON are available for many languages. For more examples comparing JSON and XML, see http://json.org/example.html.

JSON Web Services

JSON web services are web services that uses the JSON encoding format instead of, for instance, XML. Such web services can be JAX-WS web services or REST web services.
REST web services typically allow for the resource representation to be in a format selected by the client at the time of the request. Possible formats include XML, JSON etc.

Some characteristics of web services (REST and JAX-WS) that use JSON are:

- JSON text can be dynamically executed as JavaScript.
  Depending on the circumstances, this might be an asset or a risk.
- Minimizes the size of the data.
  JSON is not as verbose as XML while retaining some of the flexibility and extensibility.
- JSON web services cannot use the web security mechanisms that requires SOAP header blocks.

5.3 SOAP vs. REST Web Services

Compare SOAP web services to REST Web Services.

References:
http://java.sun.com/developer/technicalArticles/WebServices/restful/

REST web services are more appropriate when:

- The web service is completely stateless.
- A caching infrastructure is considered to increase performance and scalability.
- Service producer and consumer have a mutual understanding of the context and content produced by the service.
  This is not entirely true, since WSDL 2.0 can be used to describe a REST web service.
- Bandwidth is of importance and needs to be limited.
  By using URIs representing references to other resources, the size of the requested resource can be kept to a minimum and the client can request referenced resources as they are needed.
- Web service delivery or aggregation into existing web sites needs to be easily facilitated.
  REST web services can be consumed from different kinds of clients, such as web applications, perhaps using AJAX, plain HTML pages and other web services.
- Different client wants to retrieve resource representations in different formats, such as XML, JSON etc.
  REST web services enables the client to specify the format of the representation at request time.

SOAP-based web services are more appropriate when:

- A formal contract that describes the web service must be established.
  WSDL facilitates this.
- The architecture must address complex nonfunctional requirements.
  Examples of such requirements are: Transaction, Security, Addressing, Trust etc.
- The architecture needs to handle asynchronous processing and invocation.
  JAX-WS is able to leverage such requirements.

5.4 SOAP vs. JSON Web Services

Compare SOAP web services to JSON Web Services.

JSON web services are more appropriate when:

- Serializing simpler data structures, as opposed to documents like web pages etc.
  In the latter case XML or some other format may be more appropriate.
- Clients are AJAX clients.
- Bandwidth usage needs to be minimized.
- Validation of the format and contents of the data is not required.
  As of writing, there is a JSON schema proposal and, at least, a JavaScript implementation of a validator.

SOAP web services are more appropriate when:

- You want to ensure that the data returned by the web service can not be evaluated as JavaScript (security issue).
- The format, and to some extent the contents, of the data sent to and received from the web service needs to be validated.
  XML schema enables this.
- Better tooling is required.
- Reuse of existing data structure definitions is desired.
- Web service security involving encryption of part(s) of messages, handling different parts of a messages in different ways, end-to-end security and not only transport layer encryption.
  See section 8 for web service standards and initiatives available for SOAP web services.

5.5 JAXP APIs

Describe the functions and capabilities of the APIs included within JAXP.

References:
https://jaxp.dev.java.net/
https://jaxp-sources.dev.java.net/nonav/docs/spec/html/
http://java.sun.com/webservices/reference/tutorials/jaxp/html/docinfo.html
http://www.developer.com/xml/article.php/3397691

In this section we will take a look at the different APIs included in JAXP. Example programs for each API can be found in section 9.2.
JAXP stands for Java API for XML Processing. JAXP 1.4 supports the following technologies:

- XML
  Both XML 1.0 and XML 1.1 are supported.
  Namespaces in XML, version 1.0 and 1.1 are supported.
  XML schema is supported.
- SAX
  Simple API for XML parsing 2.0.2 is supported.
- DOM
  Document Object Model level 3 is supported.
- StAX
  Streaming API for XML is supported as a related technology, not as an endorsed specification.
- XSLT (previously TrAX)
  XML Stylesheet Language for Transformations version 1.0 is supported.
- XPath
  XPath 1.0 is supported.
- XInclude
  XML Inclusion, XInclude, version 1.0 is supported.

SAX

Tutorial: http://java.sun.com/webservices/reference/tutorials/jaxp/html/sax.html

SAX provides an event-driven, serial access mechanism that processes XML data one element at a time. SAX is a push API, which means that the client implements callbacks that are invoked as certain data, such as elements, are encountered in the document being parsed.
SAX uses less processing power and memory, but does not give access to the entire object model of an XML document. It is more suitable for server-side and high performance applications that, for instance, filters data.
SAX is oriented towards state independent processing, where the handling of an element does not depend on the element(s) that came before it.

Functions: Read XML data.
Capabilities: Generates different kinds of events when an XML document is read that can be reacted upon by handlers implemented by the user of the API.
When to use: State independent processing.

DOM

Tutorial: http://java.sun.com/webservices/reference/tutorials/jaxp/html/dom.html

DOM parses an entire XML document and creates an in-memory tree structure of objects representing the document. This approach requires more processing power and memory, compared to SAX.
DOM is mainly to be used for interactive applications that require the entire object model to be present in memory, where it can be manipulated.

Functions:	Read and write XML data.
Capabilities:	Maintain an object model of an entire document in memory. Manipulate the object model representing a document.

StAX

Tutorial: http://java.sun.com/webservices/reference/tutorials/jaxp/html/stax.html

StAX provides a streaming, event-driven, pull-parsing API for reading and writing XML documents. Pull-parsing means that the client requests data from the parser when needed. Pull-parsing, among other advantages, makes the parallel processing of multiple documents with one single thread possible. The StAX programming model is simpler than that of SAX and also has more efficient memory management than DOM.
As opposed to SAX, StAX can not only read, but also write XML data.

Functions: Read and write XML data.
Capabilities: High-performance stream filtering, processing, and modification.

XSLT

Tutorial: http://java.sun.com/webservices/reference/tutorials/jaxp/html/xslt.html

XSLT enables writing of XML data to file, converting XML data to other formats and, in conjunction with SAX, conversion of legacy data to XML.

Functions: Convert XML data to other formats.
Writing XML data to files.
Capabilities: Given a set of transformation instructions, convert data in XML format to some other format.

Comparing JAXP APIs

The following table compares the above APIs:

	StAX	SAX	DOM
API Type	Pull, streaming	Push, streaming	In memory, tree
Ease of Use	High	Medium	High
XPath Capable	No	No	Yes
CPU & Memory Efficiency	Good	Good	Varies
Forward Only	Yes	Yes	No
Reads XML	Yes	Yes	Yes
Writes XML	Yes	No	Yes
Create, Read, Update, Delete	No	No	Yes

5.6 JAXB

Describe the functions and capabilities of JAXB, including the JAXB process flow, such as XML-to-Java and Java-to-XML, and the binding and validation mechanisms provided by JAXB.

References:
https://jaxb.dev.java.net/

JAXB is short for Java Architecture for XML Binding. JAXB greatly simplifies the process of marshaling and unmarshaling XML data to/from Java objects.

JAXB Functions and Capabilities

The following things can be accomplished by JAXB 2.0:

- From an XML schema documents, generate Java interfaces and classes that represents the schema.
- From Java interfaces and classes representing an XML schema, generate an XML schema document.
- Unmarshal an XML document, and optionally validate the source data, creating a tree of content objects representing the content and organization of the XML document.
- Marshal a tree of contents objects representing the content and organization of an XML document, creating an XML document.
- Validate a tree of contents objects representing the content and organization of an XML document against an XML schema.
- Access the data of an (unmarshalled) XML document in any order desired.
- Customize, for example:
  - Binding style.
    Normally each element in a complex type is mapped to a unique content property. Alternatively, schema components that have complex types and are nested in the schema can be mapped to Java interfaces.
  - Specify names of classes to be generated.

  Binding customization can be embedded in XML schema documents or in external binding customization files.
- Marshal binary data, such as MTOM and MIME attachments.

JAXB Process Flow

For example programs, please see section 9.2 on XML Processing.

XML-to-Java

Accessing the data of an XML document using JAXB are done in the following steps:

- Bind the schema of the XML document.
  This means to generate a set of Java classes that represents the XML schema. This is accomplished using a JAXB binding compiler.
- Unmarshall the document into the Java objects.
  Optionally, the XML document can be validated against the XML schema, as part of the unmarshaling process.

Java-to-XML

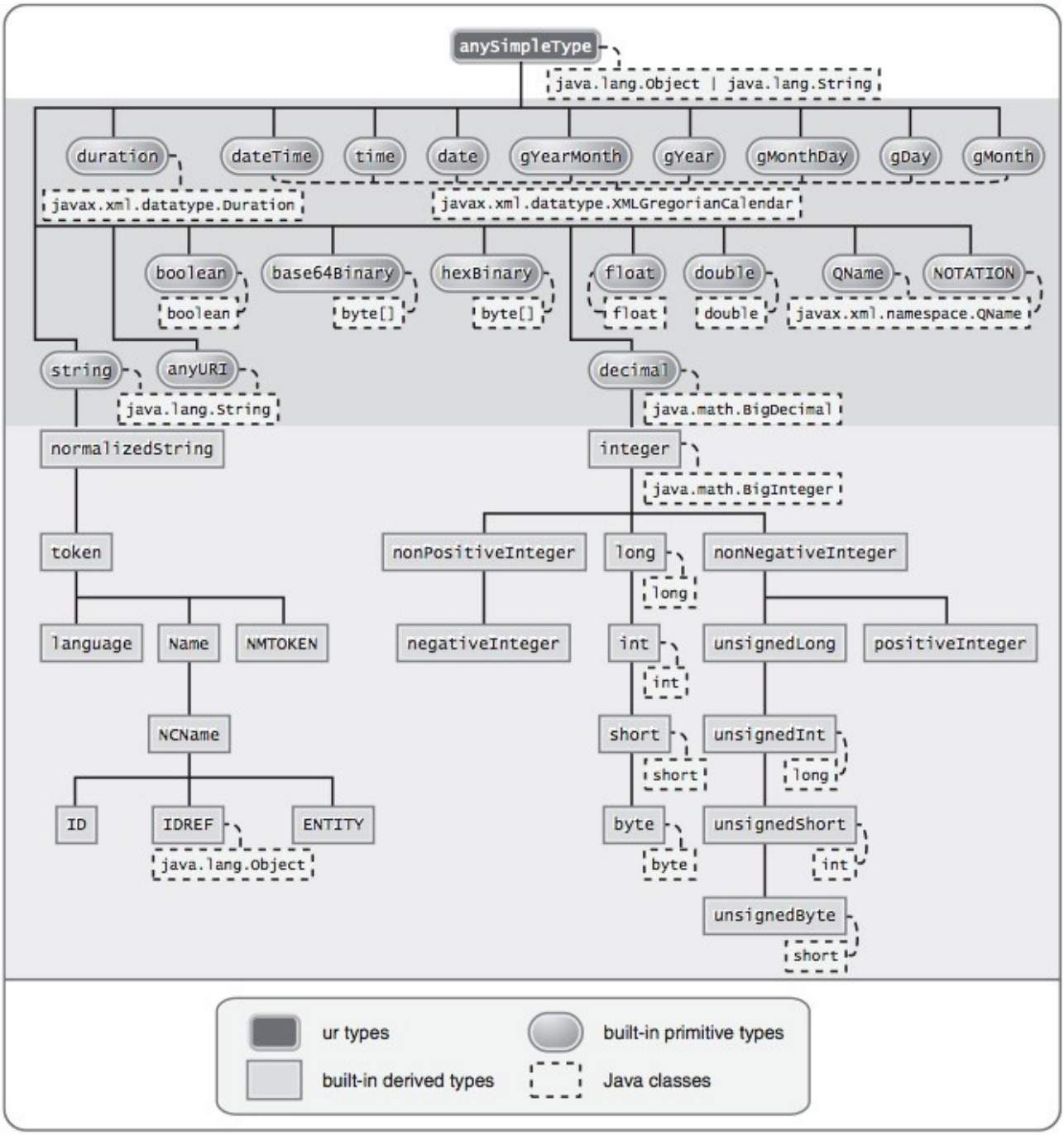
There are three basic steps when marshaling, that is, converting Java object(s) to XML:

- Bind the schema of the XML document.
- Create the Java object content tree.
- Marshal the content tree into the XML document.

JAXB Binding Mechanisms

References:
http://java.sun.com/xml/ns/jaxb/
JAXB 2.1 Specification, chapter 6 and 7

All JAXB implementations are required to implement default bindings for a set of XML schema component type. To customize the binding of an XML schema, the schema can be annotated with mapping annotations. Additionally, extensions to the binding language can be specified.
The following figure, from the JAXB specification, shows XML schema built-in atomic data types and their default mapping to Java.



It is also possible to customize bindings annotating an XML schema, as can be seen in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="unqualified"
    jaxb:extensionBindingPrefixes="xjc"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0">

    <!--
        These are global customizations which apply to all of the
        current XML schema.
    -->
    <annotation>
        <appinfo>
            <!--
                Setting this property to true will cause generated
                classes to contain a method for each property that
                allows for querying whether the property has been
                set or not.
            -->
            <jaxb:globalBindings generateIsSetMethod="true">

                <!--
                    This customization is a vendor customization which
                    is not defined by the JAXB specification.
                    The XJC customizations require the JAXB XJC compiler
                    to run in the "-extension" mode.
                    This particular customization causes generated classes
                    to implement the java.io.Serializable interface and to have
                    a serialVersionUID with the specified value.

                    <xjc:serializable uid="12343" />
                -->

                <!--
                    This customization causes all occurrences of the
                    XML schema type string to be mapped to the Java
                    type short.
                    The printMethod property specifies the method that
                    will marshal, that is convert a short value to a string
                    value and the parseMethod property specifies the method
                    that will unmarshal, that is convert a string value
                    to a short value.

                    <jaxb:javaType
                        name="short"
                        xmlType="string"
                        printMethod="javax.xml.bind.DatatypeConverter.printShort"
                        parseMethod="javax.xml.bind.DatatypeConverter.parseShort"/>
                -->
            </jaxb:globalBindings>
        </appinfo>
    </annotation>

    <element name="kompisRelation" type="ivan:KompisRelation">
        <!--
            If a JAXB binding customization is inserted here,
            like in the example below, there will be a class named
            FriendRelationElement that extends JAXBElement<KompisRelation>.

            <annotation>
                <appinfo>
                    <jaxb:class name="FriendRelationElement"/>
                </appinfo>
            </annotation>
        -->
    </element>
    <element name="person" type="ivan:Person"/>

    <attribute name="eyeColour" type="string" default="blue"/>

    <complexType name="KompisRelation">
```

```
        <!--
            If a JAXB binding customization is inserted here,
            there will be a class named FriendRelation that contains
            the properties of the KompisRelation complex type.
        -->
        <annotation>
            <appinfo>
                <jaxb:class name="FriendRelation">
                    <!--
                        This will insert the supplied JavaDoc into the
                        generated class.
                    -->
                    <jaxb:javadoc>
                        <![CDATA[Some JavaDoc for the FriendRelation class.]]>
                    </jaxb:javadoc>
                </jaxb:class>
            </appinfo>
        </annotation>

        <sequence>
            <element name="person" type="ivan:Person"/>
            <element name="friend" type="ivan:Person" minOccurs="0"
                maxOccurs="unbounded" nillable="false"/>
        </sequence>
        <attribute name="degree" type="string" use="optional"
            default="acquaintance"/>
    </complexType>

    <complexType name="Person">
        <sequence>
            <element name="firstName" type="string" nillable="false"/>
            <element name="lastName" type="string" nillable="false"/>
            <element name="age" type="int"/>
            <element name="favColour" type="string" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
        <attribute ref="ivan:eyeColour"/>
        <attribute name="hasDog" type="boolean"/>
    </complexType>
</schema>
```

Binding customizations can also be stored in a separate file, in order not to have to modify the XML schema. Extracting the binding customizations from the above XML schema into a separate file would yield the following result:

```
<jaxb:bindings
    version="2.0"
    xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc">

    <jaxb:bindings schemaLocation="kompisSchema.xsd" node="/xs:schema">
        <!--
            Setting this property to true will cause generated
            classes to contain a method for each property that
            allows for querying whether the property has been
            set or not.
        -->
        <jaxb:globalBindings generateIsSetMethod="true">

            <!--
                This customization is a vendor customization which
                is not defined by the JAXB specification.
                The XJC customizations require the JAXB XJC compiler
                to run in the "-extension" mode.
                This particular customization causes generated classes
                to implement the java.io.Serializable interface and to have
                a serialVersionUID with the specified value.
            -->
            <xjc:serializable uid="12343" />

            <!--
                This customization causes all occurrences of the
                XML schema type string to be mapped to the Java
                type short.
```

```
                The printMethod property specifies the method that
                will marshal, that is convert a short value to a string
                value and the parseMethod property specifies the method
                that will unmarshal, that is convert a string value
                to a short value.

                <jaxb:javaType
                    name="short"
                    xmlType="string"
                    printMethod="javax.xml.bind.DatatypeConverter.printShort"
                    parseMethod="javax.xml.bind.DatatypeConverter.parseShort"/>
            -->
        </jaxb:globalBindings>

        <!--
            The following binding modifications are applied to the
            complex type KompisRelation.
            Specifying the name attribute in the <jaxb:class> element
            causes the generated class to be named as specified.
        -->
        <jaxb:bindings node="//xs:complexType[@name='KompisRelation']">
            <jaxb:class name="FriendRelation">
                <jaxb:javadoc>
                    <![CDATA[Some JavaDoc for the FriendRelation class.]]>
                </jaxb:javadoc>
            </jaxb:class>
        </jaxb:bindings>

        <!--
            If a JAXB binding customization is inserted here,
            like in the example below, there will be a class named
            FriendRelationElement that extends JAXBElement<KompisRelation>.

        <jaxb:bindings node="//xs:element[@name='kompisRelation']">
            <jaxb:class name="FriendRelationElement"/>
        </jaxb:bindings>
        -->

    </jaxb:bindings>
</jaxb:bindings>
```

JAXB Validation Mechanisms

The JAXB 2.1 specification document says that there exists four forms of validation within the JAXB architecture:

- Unmarshal-time Validation
  As seen in the XML-to-Java example program above, an instance of the *Unmarshaller* class can be configured to validate the XML data as part of the unmarshalling process.
- Marshal-time Validation
  As seen in the Java-to-XML example program above, an instance of the *Marshaller* class can be configured to validate the generated XML data as part of the marshalling process.
- On-demand Validation
  This validation mode has been deprecated is JAXB 2.0.
- Fail-fast Validation
  This validation mode enables a client to receive immediate notifications when a modification to a Java object tree violates a type constraint of a Java property. Supporting this validation mode is optional in the JAXB 2.1 specification.

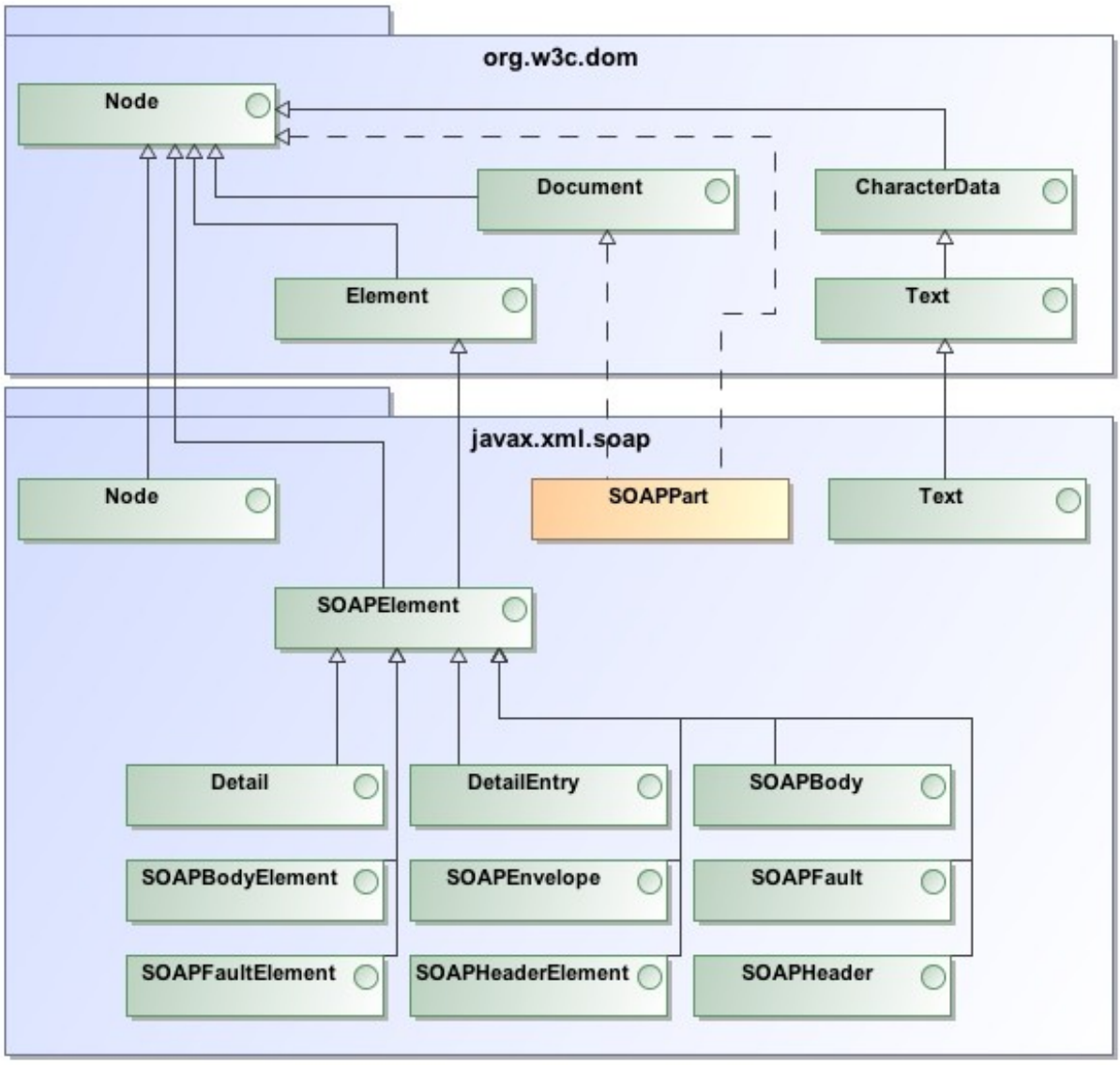
Instances of the *Unmarshaller* and *Marshaller* classes support the registration of an object that will receive any events related to validation, an object implementing the *ValidationEvent* interface.

5.7 SOAP Message with Attachment Using SAAJ

Create and use a SOAP message with attachments using the SAAJ APIs.

References:
JavaEE 5 Tutorial, chapter 19.

In the new version of SAAJ, there are relationships between some artifacts in the DOM API and artifacts in the SAAJ API, as shown in the class diagram below. This makes it possible to use the DOM API to manipulate a SOAP message created by SAAJ and vice versa.
Be careful, however - there is an example in the following SAAJ code to show that if DOM, then SAAJ and finally DOM again is used to manipulate a SOAP message, references to DOM nodes can become invalid – please consult the example code for details!



Relationships between some classes in the SAAJ API and some classes in the DOM API.

The SAAJ example program uses three additional files, as follows. The first is named "attachment.txt":

```
This file is to be attached to a SOAP message.
It contains some very important data that are to be sent to the web service.
Blah blah blah.
This is the first attachment.
```

The second file is named "attachment2.txt":

```
This is the second attachment.
```

Finally, the third file is named "soap_input.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns:ik="http://www.ivan.com/bookprice">
<!-- The body will be removed after the SOAP message has been read. -->
<soap:Body><ik:getBookPrice><isbn>0</isbn></ik:getBookPrice></soap:Body>
</soap:Envelope>
```

The SAAJ example programs are contained in two classes, the first one being a helper to output SOAP messages with attachments:

```
package com.ivan;

import java.io.ByteArrayOutputStream;
import java.io.PrintWriter;
import java.util.Iterator;
import javax.xml.soap.MimeHeader;
import javax.xml.soap.SOAPMessage;

/**
 * Class containing utility methods used with the SAAJ examples.
 *
 * @author Ivan A Krizsan
 */
public final class SAAJExampleUtils
{
    /**
     * Creates a string representation of the supplied SOAP message,
     * including MIME headers and attachments.
     *
     * @param inSOAPMsg SOAP message to create string representation of.
     */
    public static String outputSOAPMsg(final SOAPMessage inSOAPMsg)
    {
        ByteArrayOutputStream theBAOS = new ByteArrayOutputStream();
        PrintWriter theWriter = new PrintWriter(theBAOS);

        theWriter.println("MIME Headers:");
        Iterator theMimeHdrIter = inSOAPMsg.getMimeHeaders().getAllHeaders();
        for (; theMimeHdrIter.hasNext();)
        {
            MimeHeader theHdr = (MimeHeader)theMimeHdrIter.next();
            theWriter.println(theHdr.getName() + " - " + theHdr.getValue());
        }
        try
        {
            theWriter.println();
            theWriter.println("SOAP Message:");
            theWriter.flush();
            inSOAPMsg.writeTo(theBAOS);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
        return new String(theBAOS.toByteArray());
    }
}
```

The second class contains the sample programs exploring the different aspects of SAAJ. To run the different examples, the main method must be modified to invoke the appropriate example. Each public method, except for the main method and the constructor, contains a standalone example.

```
package com.ivan;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;
import java.util.Locale;

import javax.activation.DataHandler;
import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.MimeHeaders;
import javax.xml.soap.Name;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPBodyElement;
import javax.xml.soap.SOAPConstants;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.SOAPFault;
import javax.xml.soap.SOAPHeader;
import javax.xml.soap.SOAPHeaderElement;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.transform.Source;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.sax.SAXSource;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

/**
 * This class contains examples on how to use SAAJ to create and
 * manipulate SOAP messages, with and without attachments.
 *
 * @author Ivan A Krizsan
 */
public class SAAJExamples
{
    /* Constant(s): */
    private final static String SOAP_FILE = "soap_input.xml";
    private final static String ATTACHMENT_FILE_1 = "attachment.txt";
    private final static String ATTACHMENT_FILE_2 = "attachment2.txt";

    /* Instance variable(s): */
    /** Factory for creating SOAPMessage objects. */
    private MessageFactory mSOAPMsgFactory;
    /** Factory for creating objects associated with SOAP messages. */
    private SOAPFactory mSOAPObjectsFactory;

    /**
     * Application main entry-point.
     *
     * @param args Command line arguments, not used.
     */
    public static void main(String[] args)
    {
        try
        {
            SAAJExamples theInstance = new SAAJExamples();
            theInstance.createMessageWithFault();

        } catch (Exception theException)
        {
            theException.printStackTrace();
```

```
        }
    }

    /**
     * Creates an instance of this class and performs the following
     * initialization:<br/>
     * - Creates a SOAP 1.2 messages factory.<br/>
     * - Creates a factory that creates objects associated with SOAP
     * messages.<br/>
     *
     * @throws SOAPException If error occurs creating SOAP factory.
     */
    public SAAJExamples() throws SOAPException
    {
        /*
         * Create a factory for creating SOAP message objects. A
         * message factory can be configured to create SOAP 1.2
         * messages, as in the case below, or SOAP 1.1 messages or to
         * allow for both versions of SOAP messages. The latter allow
         * for processing of incoming SOAP 1.1 and 1.2 messages.
         */
        mSOAPMsgFactory =
            MessageFactory.newInstance(SOAPConstants.SOAP_1_2_PROTOCOL);

        /*
         * Create a factory that creates various objects that can
         * exist in a SOAP message.
         */
        mSOAPObjectsFactory = SOAPFactory.newInstance();
    }

    /**
     * Creates a new SOAP message and removes the SOAP <Header> from
     * the message.
     *
     * @throws SOAPException If error occurs creating SOAP message.
     */
    private void createMsgRemoveHeader() throws SOAPException
    {
        SOAPHeader theHeader;
        SOAPMessage theMsg = createEmptySOAPMessage();

        System.out.println("\nBefore:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);

        /*
         * Try to retrieve the header from the SOAP message after it
         * has been removed.
         */
        theHeader = theMsg.getSOAPHeader();
        System.out.println("\nSOAP header in the message after its removal: "
            + theHeader);

        System.out.println("\nAfter:");
        theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }

    /**
     * Creates an empty SOAP message. The code in this method also
     * demonstrates how to access the different parts of a SOAP
     * message.
     *
     * @return Empty SOAP message.
     * @throws SOAPException If error occurs creating or accessing
     * SOAP message.
     */
    public SOAPMessage createEmptySOAPMessage() throws SOAPException
    {
        SOAPMessage theMsg = mSOAPMsgFactory.createMessage();

        /*
         * Here, different ways of accessing the different parts of a
         * SOAP message are shown. First going via the SOAP part and
         * the SOAP envelope...
         */
```

```
        SOAPPart thePart = theMsg.getSOAPPart();
        SOAPEnvelope theEnvelope = thePart.getEnvelope();
        SOAPBody theBody = theEnvelope.getBody();
        SOAPHeader theHeader = theEnvelope.getHeader();

        System.out.println("The long way:");
        System.out.println(" The SOAP part: " + thePart);
        System.out.println(" The SOAP envelope: " + theEnvelope);
        System.out.println(" The SOAP body: " + theBody);
        System.out.println(" The SOAP header: " + theHeader);

        /* A faster way... */
        theBody = theMsg.getSOAPBody();
        theHeader = theMsg.getSOAPHeader();

        System.out.println("The short way:");
        System.out.println(" The SOAP body: " + theBody);
        System.out.println(" The SOAP header: " + theHeader);

        return theMsg;
    }

    /**
     * Creates an empty SOAP message. The code in this method also
     * demonstrates how to access the different parts of a SOAP
     * message.
     *
     * @return Empty SOAP message.
     * @throws SOAPException If error occurs creating or accessing
     * SOAP message.
     * @throws IOException If error occurs reading file containing
     * SOAP message.
     */
    public SOAPMessage createSOAPMessageFromFile() throws SOAPException,
        IOException
    {
        /*
         * Prepare stream reading from file containing SOAP message
         * and an object containing the MIME headers. It is also
         * possible to supply a null MimeHeaders object, in which case
         * an empty one will be created when the SOAP message is
         * created. In this example we do not include any MIME
         * headers.
         *
         * Note that if the format of the SOAP message in the file is
         * illegal, there will not be any errors until we try to get
         * some part of the SOAP message, for instance the body. To
         * see this behavior, change the protocol when retrieving the
         * message factory above to SOAP 1.1.
         */
        FileInputStream theFIS = new FileInputStream(SOAP_FILE);
        MimeHeaders theMimeHdrs = new MimeHeaders();
        SOAPMessage theMsg = mSOAPMsgFactory.createMessage(theMimeHdrs, theFIS);
        theFIS.close();

        SOAPBody theBody = theMsg.getSOAPBody();
        SOAPHeader theHeader = theMsg.getSOAPHeader();

        System.out.println("Retrieving parts of SOAP message:");
        System.out.println(" The SOAP body: " + theBody);
        System.out.println(" The SOAP header: " + theHeader);

        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);

        return theMsg;
    }

    /**
     * Creates a new SOAP message and inserts various kinds of
     * contents into the message.
     *
     * @throws SOAPException If error occurs creating SOAP message.
     */
    public void createMessageWithContent() throws SOAPException
    {
        Name theElementName;
```

```
        Name theAttributeName;
        QName theQElemName;
        SOAPBodyElement theBodyElem;
        SOAPElement theElement;
        SOAPMessage theMsg = mSOAPMsgFactory.createMessage();

        /*
         * Setting this property causes SAAJ to include an XML
         * declaration in the SOAP message. The XML declaration will
         * be for XML version 1.0 and UTF-8 encoding.
         */
        theMsg.setProperty(SOAPMessage.WRITE_XML_DECLARATION, "true");

        /* Add an element to the SOAP body. */
        theElementName =
            mSOAPObjectsFactory.createName("elemLocalName", "elemPrefix",
                "urn:elemUri");
        theBodyElem = theMsg.getSOAPBody().addBodyElement(theElementName);

        /*
         * Add a child element of the SOAP body and some contents of
         * the element. Note that SAAJ is smart enough not to repeat
         * the namespace declaration in the child element, since it is
         * the same as in the parent element.
         */
        theElementName =
            mSOAPObjectsFactory.createName("anotherElement", "elemPrefix",
                "urn:elemUri");
        theElement = theBodyElem.addChildElement(theElementName);

        /*
         * Here we use a QName to add a child element of the element
         * in the SOAP body and some contents of the element. Even
         * though the namespace having the supplied prefix is just
         * defined in a sibling element and not in the parent element,
         * SAAJ will see to that the new body element contains a
         * correct namespace definition. If the prefix has not been
         * used in the SOAP message, an exception will be thrown.
         */
        theQElemName =
            theBodyElem.createQName("bodyElemLocalName", "elemPrefix");
        theElement = theElement.addChildElement(theQElemName);
        theElement.addTextNode("Some text in an element");

        /* Add an attribute to the second body element. */
        theAttributeName = mSOAPObjectsFactory.createName("myAttribName");
        theElement.addAttribute(theAttributeName, "some attribute value");

        /* Print the SOAP message to the console. */
        System.out.println("\nThe SOAP message with content:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }

    /**
     * Creates a new SOAP message and sets its contents to that of a
     * file. The file is read using DOM or SAX, but we could also have
     * used a <code>StreamSource</code> object directly.
     *
     * @param inUseDOMFlag True if DOM is to be used to read the file
     * containing the SOAP message, false if SAX is to be used.
     * @throws Exception If error occurs reading file or creating SOAP
     * message.
     */
    public void useDOMCreateMessageFromFile(final boolean inUseDOMFlag)
        throws Exception
    {
        Source theSOAPMsgSource;
        String theMsgString;

        /* Create the SOAP message. */
        SOAPMessage theMsg = mSOAPMsgFactory.createMessage();

        if (inUseDOMFlag)
        {
            /* Use DOM to read an XML file containing the message. */
            DocumentBuilderFactory theDBFactory =
```

```
            DocumentBuilderFactory.newInstance();
        theDBFactory.setNamespaceAware(true);
        DocumentBuilder theDocumentBuilder =
            theDBFactory.newDocumentBuilder();
        File theXMLFile = new File(SOAP_FILE);
        Document theDocument = theDocumentBuilder.parse(theXMLFile);
        theSOAPMsgSource = new DOMSource(theDocument);

        theMsgString = "\nUsing DOM to read SOAP message from file:";
    } else
    {
        /*
         * SAX could also have been used to read the XML file
         * containing the message, as seen from the following code
         * snippet:
         */
        FileReader theFileReader = new FileReader(SOAP_FILE);
        InputSource theInputSource = new InputSource(theFileReader);
        theSOAPMsgSource = new SAXSource(theInputSource);

        theMsgString = "\nUsing SAX to read SOAP message from file:";
    }

    /*
     * Replace the entire SOAP part of the message with that from
     * the supplied source.
     */
    theMsg.getSOAPPart().setContent(theSOAPMsgSource);

    /* Print the SOAP message to the console. */
    System.out.println(theMsgString);
    String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
    System.out.println(theOutput);
}

/**
 * Creates a new SOAP message and sets its contents to that of a
 * file. The file is read using DOM, but we could also have used a
 * <code>StreamSource</code> object directly.
 *
 * @throws Exception If error occurs reading file or creating SOAP
 * message.
 */
public void readBodyFromSAXorDOM() throws Exception
{
    /* Create the SOAP message. */
    SOAPMessage theMsg = mSOAPMsgFactory.createMessage();

    /*
     * Use DOM to read an XML file containing the contents of the
     * SOAP body.
     */
    DocumentBuilderFactory theDBFactory =
        DocumentBuilderFactory.newInstance();
    theDBFactory.setNamespaceAware(true);
    DocumentBuilder theDocumentBuilder = theDBFactory.newDocumentBuilder();
    File theXMLFile = new File(SOAP_FILE);
    Document theDocument = theDocumentBuilder.parse(theXMLFile);

    /*
     * Replace the entire SOAP part of the message with that from
     * the supplied source.
     */
    theMsg.getSOAPBody().addDocument(theDocument);

    /* Print the SOAP message to the console. */
    System.out.println("\nUsing DOM to read SOAP message from file:");
    String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
    System.out.println(theOutput);
}

/**
 * Manipulates a SOAP message using DOM and SAAJ.
 * Shows that if first DOM is used to manipulate the message, then
 * SAAJ is used and finally an attempt to use DOM on a node obtained
 * earlier, the node reference is no longer valid.
 *
```

```
     * @throws Exception If error occurs.
     */
    public void manipulateDOM_SAAJ_DOM_Problem() throws Exception
    {
        SOAPMessage theMsg = createSOAPMessageFromFile();

        Document theDocument = theMsg.getSOAPPart();

        /* Modify the ISBN number in the request using DOM. */
        NodeList theNodeList =
            theDocument.getElementsByTagName("ik:getBookPrice");
        Node theRequestNode = theNodeList.item(0);
        theNodeList = theRequestNode.getChildNodes();
        Node theISBNNode = theNodeList.item(0);
        theISBNNode.setTextContent("DOM manipulation 1");

        /*
         * Now switch to using SAAJ to manipulate the request. First
         * get the <getBookPrice> element.
         */
        Iterator theChildIterator = theMsg.getSOAPBody().getChildElements();
        SOAPElement theElem = (SOAPElement)theChildIterator.next();

        /* Then get the <isbn> element and modify its contents. */
        QName theISBNName = new QName("isbn");
        theChildIterator = theElem.getChildElements(theISBNName);
        theElem = (SOAPElement)theChildIterator.next();
        theElem.setTextContent("SAAJ manipulation 1");

        /*
         * Finally, try to modify the contents of the <isbn> element
         * once more, using the reference to the DOM object obtained
         * earlier.
         */
        theNodeList = theRequestNode.getChildNodes();
        System.out.println("The request node (DOM) child count: "
            + theNodeList.getLength());
        if (theNodeList.getLength() != 0)
        {
            theISBNNode = theNodeList.item(0);
            theISBNNode.setTextContent("DOM manipulation 2");
        } else
        {
            /*
             * The request node is empty. This is the expected result
             * after having first used DOM, then SAAJ and finally
             * trying to use DOM again on a node obtained earlier.
             * Please refer to the SOAPElement.getChildElements()
             * method API documentation for details!
             */
            System.out.println("Having used DOM, then SAAJ and finally "
                + "going back to DOM, the request node does not have any "
                + "children and the ISBN number cannot be manipulated.");
        }

        System.out.println("\nThe manipulated SOAP message:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }

    /**
     * Creates a SOAP message with two attachments.
     *
     * @throws Exception If error occurs creating message or reading
     * data from files.
     */
    public void createMessageWithAttachments() throws Exception
    {
        SOAPMessage theMsg;
        AttachmentPart theAttachment;
        String theAttachmentData;
        File theAttachmentFile;

        theMsg = createSOAPMessageFromFile();

        /*
         * Create an attachment and set some MIME headers and the
```

```
         * contents of the attachment.
         * The type of object that can be passed to the attachment
         * when setting its content depends on the MIME type.
         */
        theAttachment = theMsg.createAttachmentPart();
        theAttachmentData = readAttachmentFile(ATTACHMENT_FILE_1);
        theAttachment.setContent(theAttachmentData, "text/plain");
        theAttachment.setContentId("attachment_1");
        theMsg.addAttachmentPart(theAttachment);

        /*
         * Create an attachment using an URL.
         * Instead of reading the contents of the attachment file,
         * the file is referred to using an URL to show another
         * way of adding attachments to a SOAP message.
         */
        theAttachmentFile = new File(ATTACHMENT_FILE_2);
        DataHandler theAttachmentDataHandler =
            new DataHandler(theAttachmentFile.toURI().toURL());
        theAttachment = theMsg.createAttachmentPart(theAttachmentDataHandler);
        theAttachment.setContentId("attachment_2");
        theMsg.addAttachmentPart(theAttachment);

        /* List the attachments of the SOAP message. */
        int theAttachmentCount = theMsg.countAttachments();
        System.out.println("Number of attachments in SOAP message: "
            + theAttachmentCount);
        Iterator theAttachmentIterator = theMsg.getAttachments();
        while (theAttachmentIterator.hasNext())
        {
            /* Retrieve data about the attachment and its contents. */
            theAttachment = (AttachmentPart)theAttachmentIterator.next();
            String theContentId = theAttachment.getContentId();
            String theContentType = theAttachment.getContentType();
            Object theContent = theAttachment.getContent();

            System.out.println("The attachment with content id " + theContentId
                + " has the content type " + theContentType
                + " and the object containing the contents is of the type: "
                + theContent.getClass());
        }

        /* Output SOAP message and attachments to console. */
        System.out.println("\nSOAP message with attachments:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }

    private String readAttachmentFile(final String inFileName)
        throws IOException
    {
        File theAttachmentFile = new File(inFileName);
        long theLength = theAttachmentFile.length();
        char theFileContents[] = new char[(int)theLength];
        FileReader theFileReader = new FileReader(theAttachmentFile);
        try
        {
            theFileReader.read(theFileContents);
        } finally
        {
            theFileReader.close();
        }
        return new String(theFileContents);
    }

    /**
     * Creates a SOAP message with a header block.
     * Shows how to use SAAJ to manipulate header blocks of SOAP messages.
     *
     * @throws SOAPException If error occurs creating or manipulating
     * SOAP message.
     */
    public void createMessageWithHeader() throws SOAPException
    {
        SOAPMessage theMsg;
        SOAPHeader theHeader;
        QName theHeaderQName;
```

```
        SOAPHeaderElement theHeaderElem;

        theMsg = mSOAPMsgFactory.createMessage();

        theHeader = theMsg.getSOAPHeader();
        theHeaderQName =
            new QName("http://www.ivan.com/headers", "myHeaderElem", "ikh");
        theHeaderElem = theHeader.addHeaderElement(theHeaderQName);

        /*
         * The actor attribute has been renamed to role in SOAP 1.2.
         * If one does not know whether the SOAP message is version 1.1
         * or 1.2, then use the setActor method.
         * If the setRole method is used on a SOAP 1.1 message, an exception
         * will be thrown.
         */
        theHeaderElem.setActor("urn:someActorURI");
        theHeaderElem.setRole("urn:someRoleURI");
        theHeaderElem.setMustUnderstand(true);
        theHeaderElem.setRelay(true);

        /*
         * Retrieve all headers with a specified role.
         * Similarly, it is possible to retrieve all headers that
         * must be understood.
         */
        System.out.println("\nListing header elements with common role:");
        Iterator theHeaderIterator =
            theHeader.examineHeaderElements("urn:someRoleURI");
        while (theHeaderIterator.hasNext())
        {
            theHeaderElem = (SOAPHeaderElement)theHeaderIterator.next();
            System.out.println("A header: " + theHeaderElem
                + ", must understand: " + theHeaderElem.getMustUnderstand()
                + ", relay: " + theHeaderElem.getRelay() + ", role: "
                + theHeaderElem.getActor());
        }

        /* Output SOAP message to console. */
        System.out.println("\nSOAP message with header:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }

    /**
     * Creates a SOAP message that contains a fault.
     *
     * @throws SOAPException If error occurs creating or populating
     * message.
     */
    public void createMessageWithFault() throws SOAPException
    {
        SOAPMessage theMsg;
        SOAPBody theBody;
        QName theFaultCodeQName;
        SOAPFault theFault;

        /* Create a SOAP message that will have an XML declaration. */
        theMsg = mSOAPMsgFactory.createMessage();
        theMsg.setProperty(SOAPMessage.WRITE_XML_DECLARATION, "true");

        theBody = theMsg.getSOAPBody();

        /* Create the fault message. */
        theFault = theBody.addFault();

        /* Set the fault properties - note that this is a SOAP 1.2 fault! */
        theFaultCodeQName =
            new QName(SOAPConstants.URI_NS_SOAP_1_2_ENVELOPE, "Receiver");
        theFault.setFaultCode(theFaultCodeQName);
        theFault.setFaultRole("http://www.ivan.com/some-role");

        /* Append a fault subcode. */
        theFaultCodeQName =
            new QName("http://www.ivan.com/faults", "myFaultSubcode", "ikf");
        theFault.appendFaultSubcode(theFaultCodeQName);
```

```
        /* Add two fault reason texts in different languages. */
        theFault.addFaultReasonText("The whole shebang has exploded", Locale.ENGLISH);
        theFault.addFaultReasonText("Alles ist kaputt", Locale.GERMAN);

        /* Output SOAP message to console. */
        System.out.println("\nSOAP message with header:");
        String theOutput = SAAJExampleUtils.outputSOAPMsg(theMsg);
        System.out.println(theOutput);
    }
}
```

6. JAXR

6.1 JAXR Basics

Describe the function of JAXR in Web service architectural model, the two basic levels of business registry functionality supported by JAXR, and the function of the basic JAXR business objects and how they map to the UDDI data structures.

JAXR can be used in conjunction with other kinds of registries, but in this document only UDDI registries will be considered. The motivation to this follows below.

JAXR in Web Service Architecture

References:
http://java.sun.com/blueprints/patterns/ServiceLocator.html
JAXR 1.0 Specification, section 2.1

The role of JAXR in the web service architectural model is very similar to that of a service locator (see reference above). JAXR connects to a registry, in order to:

- Publish web services.
- Discover web services.
- Utilize web services.

Using JAXR and a registry have the following advantages:

- Clients of web services does not need to maintain the location of each service they use, but can look these up dynamically.
- Can support failover.
  If a service fails, a second lookup can locate additional services that can take its place.
- Looser coupling between services and clients.

Business Registry Functionality Levels

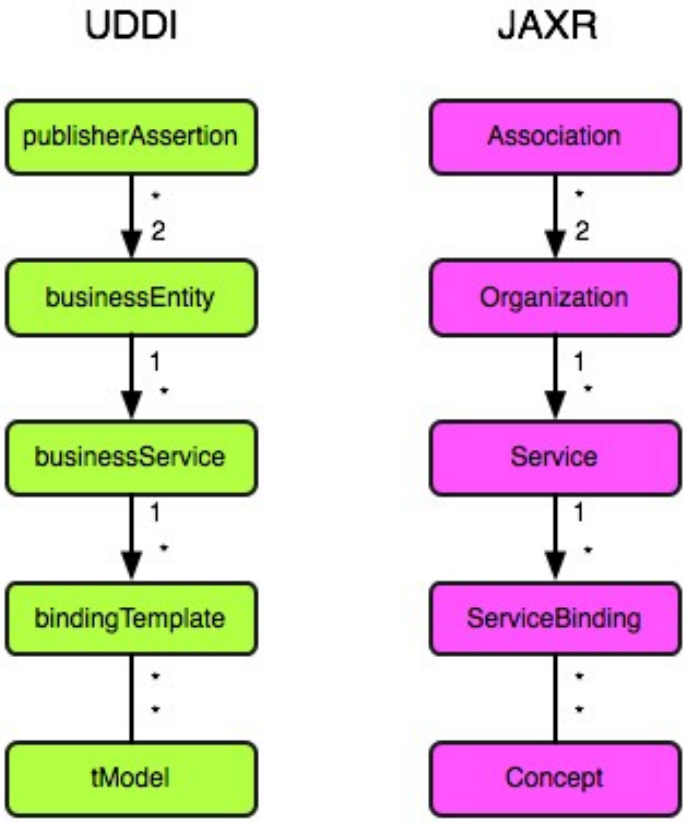
The JAXR specification mandates categorization of the methods in the JAXR API by, currently, two levels; level 0 and level 1. Such a level is called a capability profile.

- Level 0
  Level 0 is the business-level API.
  Support for the level 0 capability profile is required by all JAXR providers.
  Methods assigned to this level provides the most basic registry capabilities.
  JAXR providers for UDDI must be level 0 compliant.
- Level 1
  Level 1 is the low-level, generic, API.
  Support for the level 1 profile is optional for JAXR providers.
  Methods assigned to this level provides more advanced registry capabilities required by more demanding JAXR clients.

The WS-I Basic Profile only sanctions the use of UDDI registries. A consequence of this is that we are only allowed to use the JAXR level 0 capability profile.
Note that, when looking at classes in the JAXR API documentation, each method has a capability level specified.

JAXR Business Objects

The JAXR business objects are those objects that model business concepts. This picture shows how the most significant JAXR object types and UDDI data structures correspond:



The *RegistryObject* interface specifies common properties for most of the objects in the JAXR model.

The following table lists the most significant JAXR objects, describes their function and how they correspond to UDDI data structures.

JAXR Business Object	UDDI Data Structure	JAXR Function
Association	publisherAssertion	Represents an association between two organizations.
Organization	businessEntity	Represents an organization, such as a company, which can have the following relations: - Child organizations, any number. - Services, any number. - Users, any number.
Service	businessService	Represents a service provided by an organization, which can have the following relations: - Service bindings, any number.
ServiceBinding	bindingTemplate	Represents technical information on how to access a service. Has the following relations: - Specification links, any number. - Access URI, one.
Concept	tModel	Represent taxonomy elements and their structural relationship with each other.

6.2 JAXR Client Development

Create JAXR client to connect to a UDDI business registry, execute queries to locate services that meet specific requirements, and publish or update information about a business service.

For the examples in this section, I installed and used the Apache jUDDI v2 business registry in a local Tomcat installation and a local MySQL database.
The code below shows how to use JAXR to accomplish the following:

- Connect to a UDDI registry.
- Locate services meeting specific criteria.
- Publish information about a business service.
- Modify information about a business service.

```
package com.ivan.jaxr;

import java.net.PasswordAuthentication;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;

import javax.xml.registry.BulkResponse;
import javax.xml.registry.BusinessLifeCycleManager;
import javax.xml.registry.BusinessQueryManager;
import javax.xml.registry.CapabilityProfile;
import javax.xml.registry.Connection;
import javax.xml.registry.ConnectionFactory;
import javax.xml.registry.FindQualifier;
import javax.xml.registry.JAXRException;
import javax.xml.registry.LifeCycleManager;
import javax.xml.registry.RegistryService;
import javax.xml.registry.infomodel.Classification;
import javax.xml.registry.infomodel.ClassificationScheme;
import javax.xml.registry.infomodel.EmailAddress;
import javax.xml.registry.infomodel.ExternalLink;
import javax.xml.registry.infomodel.InternationalString;
import javax.xml.registry.infomodel.Key;
import javax.xml.registry.infomodel.Organization;
import javax.xml.registry.infomodel.PersonName;
import javax.xml.registry.infomodel.PostalAddress;
import javax.xml.registry.infomodel.RegistryObject;
import javax.xml.registry.infomodel.Service;
import javax.xml.registry.infomodel.ServiceBinding;
import javax.xml.registry.infomodel.TelephoneNumber;
import javax.xml.registry.infomodel.User;

/**
 * This class contains JAXR examples doing the following:
 * - Connect to a UDDI registry.
 * - Locate services meeting specific criteria.
 * - Publish information about a business service.
 * - Modify information about a business service.
 *
 * Configured to use a jUDDI instance running in Tomcat on localhost.
 * Note that a publisher must have been inserted into the database
 * and the appropriate constant below set to reflect the publisher id.
 *
 * Requires the following libraries:
 * jaxr-api.jar, jaxr-impl.jar, jaxb-api.jar, jaxb-impl.jar,
 * jaxb-xjc.jar, jaxb1-impl.jar, saaj-api.jar, saaj-impl.jar,
 * activation.jar
 *
 * @author Ivan A Krizsan
 */
public class JAXRExamples
{
    /* Constant(s): */
    /** JAXR built-in classification: North American industry classification system. */
    private static final String NORTH_AMERICA_INDUSTRY_BUILTIN_TAXONOMY =
```

```
        "ntis-gov:naics";
    /** JAXR built-in classification taxonomy. */
    private static final String UDDI_ORG_TYPES_BUILTIN_TAXONOMY =
        "uddi-org:types";
    /** WSDL classification as defined by UDDI standard. */
    private static final String WSDL_CLASSIFICATION = "wsdlSpec";
    private static final String PUBLICATION_MANAGER_URL =
        "http://localhost:8080/juddi/publish";
    private static final String INQUIRY_MANAGER_URL =
        "http://localhost:8080/juddi/inquiry";
    /** UDDI registry user name. */
    private static final String JUDDI_PUBLISHER_ID = "juddi";

    private static final String CALCULATOR_SERVICE_DESCR =
        "This is the description of my calculator service";
    private static final String ORGANIZATION_NAME_ACME = "Acme Inc";
    private static final String CALCULATOR_SERVICE_NAME = "Calculator Service";
    private static final String MY_ORG_CLASSIFICATION_VALUE = "463812";
    private static final String MY_ORG_CLASSIFICATION_NAME =
        "Alchoholic Beverages and Drugs";

    /* Instance variable(s): */
    private BusinessQueryManager mBusinessQueryMgr;
    private BusinessLifeCycleManager mBusinessLifecycleMgr;
    private Connection mConnection;

    /**
     * Prepares for access to the UDDI registry.
     */
    public void connectToRegistry() throws JAXRException
    {
        /*
         * Retrieve a factory that creates JAXR connections.
         * A connection represents a session with a registry provider
         * and maintains state for the session.
         */
        ConnectionFactory theJAXRConnectionFactory =
            ConnectionFactory.newInstance();

        /*
         * Set the inquiry and publication manager's URLs to be used
         * by the connection factory.
         */
        Properties theUDDIConnectionProperties = new Properties();
        theUDDIConnectionProperties.setProperty(
            "javax.xml.registry.queryManagerURL", INQUIRY_MANAGER_URL);
        theUDDIConnectionProperties.setProperty(
            "javax.xml.registry.lifeCycleManagerURL", PUBLICATION_MANAGER_URL);

        theJAXRConnectionFactory.setProperties(theUDDIConnectionProperties);

        /* Finally we can retrieve a JAXR connection. */
        mConnection = theJAXRConnectionFactory.createConnection();

        mConnection.setSynchronous(true);

        /*
         * Set the credentials for the connection.
         * JAXR will help us retrieve an UDDI authorization token and
         * include it in subsequent requests.
         * JUDDI will, in its default configuration, authenticate users
         * which are present in the PUBLISHER table, PUBLISHER_ID column
         * regardless of the password supplied.
         */
        PasswordAuthentication thePswAuthentication =
            new PasswordAuthentication(JUDDI_PUBLISHER_ID, "".toCharArray());
        Set<PasswordAuthentication> theConnectionCredentials =
            new HashSet<PasswordAuthentication>();
        theConnectionCredentials.add(thePswAuthentication);
        mConnection.setCredentials(theConnectionCredentials);

        /*
         * Get the RegistryService, which is used to retrieve,
         * among other things:
         * - BusinessLifeCycleManager which allows for creation and
         *   deletion of different kinds of objects, such as
         *   organizations, services etc.
```

```
         * - BusinessQueryManager which allows for searching among
         *   different kinds of objects.
         * - A CapabilityProfile object from which the capability
         *   level and JAXR version can be retrieved.
         */
        RegistryService theRegistryService = mConnection.getRegistryService();

        mBusinessQueryMgr = theRegistryService.getBusinessQueryManager();
        mBusinessLifecycleMgr =
            theRegistryService.getBusinessLifeCycleManager();
        CapabilityProfile theProfile =
            theRegistryService.getCapabilityProfile();

        /* Output some information about the capability of the registry. */
        int theLevel = theProfile.getCapabilityLevel();
        String theVersion = theProfile.getVersion();
        System.out
            .println("Contact with registry established, "
                + "capability level: " + theLevel + ", JAXR version: "
                + theVersion);
    }

    /**
     * Program entry point.
     *
     * @param args Command line arguments, not used.
     */
    public static void main(String[] args)
    {
        JAXRExamples theInstance = new JAXRExamples();

        try
        {
            theInstance.connectToRegistry();

            //theInstance.deleteOrganizations();
            theInstance.createMyClassificationScheme();
            theInstance.addOrganization("Acme Inc");
            theInstance.listAllOrganizations();
            theInstance.listAllOrganizations2();
            theInstance.publishService();
            theInstance.listAllServices();
            theInstance.modifyService();
            theInstance.listAllServices();

        } catch (JAXRException theException)
        {
            theException.printStackTrace();
        } finally
        {
            try
            {
                theInstance.mConnection.close();
            } catch (JAXRException theException)
            {
                /* Ignore exceptions. */
            }
        }
    }

    /**
     * Queries the UDDI registry for all organizations which name
     * contains the supplied name fragment.
     * If the name fragment supplied is the empty string, then all
     * registered organizations will be matched.
     * Querying does not require the setting of user credentials.
     *
     * @param inNameFragment Name fragment to match.
     * @return All found organizations, or null if no organization
     * found or error occurred.
     * @throws JAXRException If error occurs querying registry.
     */
    public Collection<Organization> findOrganizationsByName(
        final String inNameFragment) throws JAXRException
    {
        /*
         * Create list of query qualifiers. In this case:
```

```
         * - Case sensitive matching of item names.
         * - Sort found items by name in descending order.
         */
        ArrayList<String> theQueryQualifiers = new ArrayList<String>();
        theQueryQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
        theQueryQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

        /*
         * Create list of item names to search for.
         * The % is a wildcard character.
         */
        ArrayList<String> theNamePatterns = new ArrayList<String>();
        theNamePatterns.add("%" + inNameFragment + "%");

        /* Query for the organization(s). */
        BulkResponse theQueryResponse =
            mBusinessQueryMgr.findOrganizations(theQueryQualifiers,
                theNamePatterns, null, null, null, null);

        Collection theQueryExceptions = theQueryResponse.getExceptions();
        Collection<Organization> theOrganizations =
            theQueryResponse.getCollection();

        return theOrganizations;
    }

    /**
     * Queries the UDDI registry for all organizations which have
     * been classified by the supplied classification name and value.
     * Querying does not require the setting of user credentials.
     *
     * @param inClassificationName Classification name to match.
     * @param inClassificationValue Classification value to match.
     * @return All matching organizations, or null if no organization
     * found or error occurred.
     * @throws JAXRException If error occurs querying registry.
     */
    public Collection<Organization> findOrganizationByClassification(
        final String inClassificationName, final String inClassificationValue)
        throws JAXRException
    {
        /*
         * Obtain a parent classification scheme; one of the built-in
         * standard taxonomies of the JAXR provider.
         */
        ClassificationScheme theParentCS =
            mBusinessQueryMgr.findClassificationSchemeByName(null,
                NORTH_AMERICA_INDUSTRY_BUILTIN_TAXONOMY);

        /*
         * Create a new classification that we will use as a find
         * criteria when searching for organizations.
         */
        InternationalString theNameIS =
            mBusinessLifecycleMgr
                .createInternationalString(inClassificationName);
        Classification theFindClassification =
            mBusinessLifecycleMgr.createClassification(theParentCS, theNameIS,
                inClassificationValue);
        Collection<Classification> theFindClassifications =
            new ArrayList<Classification>();
        theFindClassifications.add(theFindClassification);

        /* Query for the organization(s). */
        BulkResponse theResponse =
            mBusinessQueryMgr.findOrganizations(null, null,
                theFindClassifications, null, null, null);
        Collection<Organization> theOrganizations = theResponse.getCollection();

        return theOrganizations;
    }

    /**
     * Lists all organizations in the repository to the console.
     */
    public void listAllOrganizations()
    {
```

```
        try
        {
            Collection<Organization> theOrganizations =
                findOrganizationsByName("");

            System.out.println("\nSearching for organizations by name:");
            printOrganizations(theOrganizations);
        } catch (JAXRException theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Lists all organizations in the repository to the console.
     */
    public void listAllOrganizations2()
    {
        try
        {
            Collection<Organization> theOrganizations =
                findOrganizationByClassification(MY_ORG_CLASSIFICATION_NAME,
                    MY_ORG_CLASSIFICATION_VALUE);

            System.out
                .println("\nSearching for organizations by classification:");
            printOrganizations(theOrganizations);
        } catch (JAXRException theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Lists information on the supplied organizations to the console.
     * This method does not interact with any registry, just outputs
     * supplied data.
     *
     * @param inOrganizations Organizations to list.
     * @throws JAXRException If error occurs retrieving organization data.
     */
    private void printOrganizations(
        final Collection<Organization> inOrganizations) throws JAXRException
    {
        System.out.println("Listing organizations:");

        if (inOrganizations != null)
        {
            /* Output the result to the console. */
            for (Organization theOrganization : inOrganizations)
            {
                User thePrimaryContact = theOrganization.getPrimaryContact();

                System.out.println("An organization: "
                    + theOrganization.getName().getValue());
                if (thePrimaryContact != null)
                {
                    Collection<PostalAddress> theAddresses =
                        thePrimaryContact.getPostalAddresses();
                    Collection<EmailAddress> theEmailAddresses =
                        thePrimaryContact.getEmailAddresses();
                    Collection<TelephoneNumber> thePhoneNumbers =
                        thePrimaryContact.getTelephoneNumbers(null);

                    /* Primary contact name, phone and email. */
                    System.out.println("   Primary contact: "
                        + thePrimaryContact.getPersonName().getFullName());
                    System.out.println("   Primary contact phone: ");
                    for (TelephoneNumber theTelephoneNumber : thePhoneNumbers)
                    {
                        System.out.println("      "
                            + theTelephoneNumber.getNumber());
                    }
                    System.out.println("   Primary contact email: ");
                    for (EmailAddress theEmailAddress : theEmailAddresses)
                    {
                        System.out.println("      "
```

```
                        + theEmailAddress.getAddress());
                }

                /* List organization's services and bindings. */
                Collection<Service> theOrgServices =
                    theOrganization.getServices();
                System.out.println("   Organization's services:");
                for (Service theService : theOrgServices)
                {
                    String theServiceName = theService.getName().getValue();
                    String theServiceDesc =
                        theService.getDescription().getValue();
                    System.out.println("      " + theServiceName + " - "
                        + theServiceDesc);

                    Collection<ServiceBinding> theOrgBindings =
                        theService.getServiceBindings();
                    System.out.println("      Service bindings:");
                    for (ServiceBinding theServiceBinding : theOrgBindings)
                    {
                        String theBindingName =
                            theServiceBinding.getName().getValue();
                        String theBindingDesc =
                            theServiceBinding.getDescription().getValue();
                        String theBindingURI =
                            theServiceBinding.getAccessURI();
                        System.out.println("         " + theBindingName
                            + " - " + theBindingDesc + " - "
                            + theBindingURI);
                    }
                }
            }
        }
    } else
    {
        System.out.println("An error occurred querying for organizations.");
    }
}

/**
 * Deletes all organizations in the registry and objects
 * owned by the organizations.
 * This operation requires setting of user credentials.
 *
 * @throws JAXRException If error occurs querying registry.
 */
public void deleteOrganizations() throws JAXRException
{
    /* No query qualifiers. */
    ArrayList<String> theQueryQualifiers = new ArrayList<String>();

    /*
     * Create list of item names to search for.
     * The '%%' means that any name will match.
     */
    ArrayList<String> theNames = new ArrayList<String>();
    theNames.add("%%");

    /* Use the business query manager to query for the organization(s). */
    BulkResponse theQueryResponse =
        mBusinessQueryMgr.findOrganizations(theQueryQualifiers, theNames,
            null, null, null, null);

    Collection theQueryExceptions = theQueryResponse.getExceptions();
    Collection<Organization> theOrganizations =
        theQueryResponse.getCollection();

    if (theQueryExceptions == null)
    {
        /*
         * Get keys for all the organizations, in order to be
         * able to specify which organizations to delete.
         */
        Collection<Key> theOrgKeys = new ArrayList<Key>();
        for (Organization theOrganization : theOrganizations)
        {
            theOrgKeys.add(theOrganization.getKey());
```

```
            }

            /* Finally we can delete all the organizations. */
            mBusinessLifecycleMgr.deleteOrganizations(theOrgKeys);

            System.out.println("Successfully deleted all organizations!");
        } else
        {
            System.out.println("An error occurred querying for organizations.");
        }
    }

    /**
     * Adds an organization with the supplied name to the registry.
     * Also creates a primary contact for the organization.
     * This operation requires setting of user credentials.
     *
     * @param inOrgName Name of organization to add.
     * @return The added organization.
     * @throws JAXRException If error occurred inserting organization
     * into registry.
     */
    public Organization addOrganization(final String inOrgName)
        throws JAXRException
    {
        /* Create an organization object and give it a name. */
        Organization theOrganization =
            mBusinessLifecycleMgr.createOrganization(inOrgName);

        /* Create the organization's primary contact. */
        User thePrimaryContact = mBusinessLifecycleMgr.createUser();
        PersonName thePersonName =
            mBusinessLifecycleMgr.createPersonName("Steven Segal");
        thePrimaryContact.setPersonName(thePersonName);

        /* Set phone number of the primary contact. */
        TelephoneNumber thePhoneNumber =
            mBusinessLifecycleMgr.createTelephoneNumber();
        thePhoneNumber.setNumber("555-123-1234");
        Collection<TelephoneNumber> thePhoneNoCollection =
            new ArrayList<TelephoneNumber>();
        thePhoneNoCollection.add(thePhoneNumber);
        thePrimaryContact.setTelephoneNumbers(thePhoneNoCollection);

        /* Set email address of the primary contact. */
        EmailAddress theEmailAddress =
            mBusinessLifecycleMgr.createEmailAddress("steven.segal@acme.com");
        Collection<EmailAddress> theEmailCollection =
            new ArrayList<EmailAddress>();
        theEmailCollection.add(theEmailAddress);
        thePrimaryContact.setEmailAddresses(theEmailCollection);

        /* Finally we can set the organization's primary contact. */
        theOrganization.setPrimaryContact(thePrimaryContact);

        /* Place all organizations to be added in a collection. */
        Collection<Organization> theOrgCollection =
            new ArrayList<Organization>();
        theOrgCollection.add(theOrganization);

        /* Classify the organization. */
        ClassificationScheme theNaicsClassificationScheme =
            mBusinessQueryMgr.findClassificationSchemeByName(null,
                NORTH_AMERICA_INDUSTRY_BUILTIN_TAXONOMY);
        Classification theOrganizationClassification =
            mBusinessLifecycleMgr.createClassification(
                theNaicsClassificationScheme, MY_ORG_CLASSIFICATION_NAME,
                MY_ORG_CLASSIFICATION_VALUE);

        theOrganization.addClassification(theOrganizationClassification);

        /* Save the organization(s) in the registry. */
        BulkResponse theQueryResponse =
            mBusinessLifecycleMgr.saveOrganizations(theOrgCollection);

        /*
         * If no exceptions in the response, then the operation
```

```
         * succeeded and there will be a key for the saved organization.
         */
        if (theQueryResponse.getExceptions() == null)
        {
            Collection<Key> theResponseCollection =
                theQueryResponse.getCollection();
            Key theOrgKey = theResponseCollection.iterator().next();
            System.out
                .println("Successfully created and saved an organization.");
            System.out.println("Organization Key = " + theOrgKey.getId());
        } else
        {
            System.out.println("An error occurred adding the organization: "
                + inOrgName);
        }

        return theOrganization;
    }

    /**
     * Publishes the calculator service belonging to the Acme organization.
     * If the Acme organization does not not exist, it will be created.
     * This operation requires setting of user credentials.
     *
     * @throws JAXRException If error occurs publishing service.
     */
    public void publishService() throws JAXRException
    {
        Collection<Organization> theOrgs;
        Collection<RegistryObject> theRegistryObjects =
            new ArrayList<RegistryObject>();
        Organization theServiceOrg;
        InternationalString theServiceDescr =
            mBusinessLifecycleMgr
                .createInternationalString(CALCULATOR_SERVICE_DESCR);

        /* Make sure we have an organization to add the service to. */
        theOrgs = findOrganizationsByName(ORGANIZATION_NAME_ACME);
        if (theOrgs.isEmpty())
        {
            theServiceOrg = addOrganization(ORGANIZATION_NAME_ACME);
        } else
        {
            theServiceOrg = theOrgs.iterator().next();
        }

        /*
         * Create a new service that the organization is to publish.
         * Here, other properties of the service could be set, such as
         * a description, categorization etc.
         */
        Service theService =
            mBusinessLifecycleMgr.createService(CALCULATOR_SERVICE_NAME);
        theService.setDescription(theServiceDescr);

        /*
         * Add a service binding to the service just created.
         * Note that URI validation is turned off, to allow us
         * to publish a fictious service URL.
         */
        ServiceBinding theServiceBinding =
            mBusinessLifecycleMgr.createServiceBinding();
        InternationalString theIS =
            mBusinessLifecycleMgr
                .createInternationalString("My service binding description");
        theServiceBinding.setDescription(theIS);
        theIS =
            mBusinessLifecycleMgr
                .createInternationalString("MyServiceBindingName");
        theServiceBinding.setName(theIS);
        theServiceBinding.setValidateURI(false);
        theServiceBinding.setAccessURI("http://www.ivan.com:8080/someservice/");

        theService.addServiceBinding(theServiceBinding);

        /* Link the service to the organization that provides the service. */
        theService.setProvidingOrganization(theServiceOrg);
```

```
        /*
         * Save the services and update the organization.
         * Note that updating both kinds of objects can be done with
         * one single call to the business lifecycle manager.
         */
        theRegistryObjects.add(theService);
        theRegistryObjects.add(theServiceOrg);
        mBusinessLifecycleMgr.saveObjects(theRegistryObjects);

        System.out.println("Successfully published a service.");
    }

    /**
     * Finds services with names matching supplied name fragment.
     * If the name fragment is the empty string, then all services will
     * be matched.
     * Querying does not require the setting of user credentials.
     *
     * @param inNameFragment Name fragment to match.
     * @return Matching services.
     * @throws JAXRException If error occurs finding services.
     */
    public Collection<Service> findServicesByName(final String inNameFragment)
        throws JAXRException
    {
        /*
         * Create list of query qualifiers. In this case:
         * - Case sensitive matching of item names.
         * - Sort found items by name in descending order.
         */
        ArrayList<String> theQueryQualifiers = new ArrayList<String>();
        theQueryQualifiers.add(FindQualifier.CASE_SENSITIVE_MATCH);
        theQueryQualifiers.add(FindQualifier.SORT_BY_NAME_DESC);

        /*
         * Create list of item names to search for.
         * The % is a wildcard character.
         */
        ArrayList<String> theNamePatterns = new ArrayList<String>();
        theNamePatterns.add("%" + inNameFragment + "%");

        /*
         * Use the business query manager to query for the services.
         * Two additional collections may be supplied as search criteria
         * for services:
         * - Collection<Classification> - Classifications that matching
         *   services must match.
         * - Collection<Concept> - Concepts that matching services must
         *   match.
         */
        BulkResponse theQueryResponse =
            mBusinessQueryMgr.findServices(null, theQueryQualifiers,
                theNamePatterns, null, null);

        Collection theQueryExceptions = theQueryResponse.getExceptions();
        Collection<Service> theServices = theQueryResponse.getCollection();

        return theServices;
    }

    /**
     * Lists all services in the repository to the console.
     */
    public void listAllServices()
    {
        try
        {
            Collection<Service> theServices = findServicesByName("");

            System.out.println("Listing services:");

            if (theServices != null)
            {
                /* Output the result to the console. */
                for (Service theService : theServices)
                {
```

```
                    System.out.println("A service: "
                        + theService.getName().getValue() + ", description: "
                        + theService.getDescription().getValue());

                    Collection<ServiceBinding> theBindings =
                        theService.getServiceBindings();
                    for (ServiceBinding theServiceBinding : theBindings)
                    {
                        System.out.println("   A binding: " +
                            theServiceBinding.getName().getValue() +
                            " - " + theServiceBinding.getAccessURI());
                    }
                }
            } else
            {
                System.out.println("An error occurred querying for services.");
            }
        } catch (JAXRException theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Creates a classification scheme.
     * This operation requires setting of user credentials.
     *
     * @throws JAXRException If error occurs.
     */
    public void createMyClassificationScheme() throws JAXRException
    {
        /* Create my WSDL service classification scheme. */
        InternationalString theSchemeName =
            mBusinessLifecycleMgr
                .createInternationalString("WSDLClassificationScheme");
        InternationalString theSchemeDescription =
            mBusinessLifecycleMgr
                .createInternationalString("WSDL Classification Scheme");
        ClassificationScheme theScheme =
            mBusinessLifecycleMgr.createClassificationScheme(theSchemeName,
                theSchemeDescription);

        /*
         * Obtain a parent classification scheme; one of the built-in
         * standard taxonomies of the JAXR provider.
         */
        ClassificationScheme theParentClassificationScheme =
            mBusinessQueryMgr.findClassificationSchemeByName(null,
                NORTH_AMERICA_INDUSTRY_BUILTIN_TAXONOMY);

        /* Create my very own WSDL service classification. */
        Classification theWSDLClassification =
            mBusinessLifecycleMgr.createClassification(
                theParentClassificationScheme, WSDL_CLASSIFICATION,
                WSDL_CLASSIFICATION);

        theScheme.addClassification(theWSDLClassification);

        /*
         * Set fictitious link to location of scheme so it can be
         * looked up. If the URI were valid, the createExternalLink method
         * could have been used.
         */
        ExternalLink theExtLink =
            (ExternalLink)mBusinessLifecycleMgr
                .createObject(LifeCycleManager.EXTERNAL_LINK);
        theExtLink.setValidateURI(false);
        theExtLink.setExternalURI("http://www.ivan.com/myservicescheme");

        InternationalString theLinkDescr =
            mBusinessLifecycleMgr
                .createInternationalString("Description of link to my services scheme");
        theExtLink.setDescription(theLinkDescr);
        theScheme.addExternalLink(theExtLink);

        /* Write the scheme to the registry. */
        Collection<ClassificationScheme> theSchemesCollection =
```

```
            new ArrayList<ClassificationScheme>();
        theSchemesCollection.add(theScheme);

        BulkResponse theBulkResponse =
            mBusinessLifecycleMgr
                .saveClassificationSchemes(theSchemesCollection);

        System.out.println("Successfully created classification scheme.");
    }

    /**
     * Modifies the Calculator Service binding URL.
     * The new URL will have the current time in milliseconds appended,
     * so it will change for every invocation of this method.
     *
     * @throws JAXRException If error occurs modifying service.
     */
    public void modifyService() throws JAXRException
    {
        Collection<Service> theServices;
        Service theCalcService;

        /* Find the service to modify. */
        theServices = findServicesByName("Calculator Service");
        if (theServices.isEmpty())
        {
            System.out.println("Calculator Service not found, cannot modify.");
            return;
        }

        /* Service to modify is the first one in the collection. */
        theCalcService = theServices.iterator().next();

        /* Modify the URL of the first servce binding. */
        Collection<ServiceBinding> theBindings =
            theCalcService.getServiceBindings();
        ServiceBinding theBinding = theBindings.iterator().next();

        theBinding.setAccessURI("http://www.navi.com/newservice" +
            System.currentTimeMillis() + "/");

        /* Save the updated service. */
        mBusinessLifecycleMgr.saveServices(theServices);
    }
}
```

7. Java EE Web Services

7.1 APIs Characteristics and Services

Identify the characteristics of, and the services and APIs included in, the Java EE platform.

References:
http://java.sun.com/javaee/overview/faq/
http://java.sun.com/javaee/technologies/

Characteristics of the Java EE Platform

Some characteristics of the Java EE platform are:

- Industry standard for developing Java server-side applications.
- In addition to Java SE, the Java EE platform provides:
  - Web services.
  - A component model.
  - Management and communication APIs.
- Provides a framework for developing and deploying web services.
- Uses containers that provide enterprise infrastructure.
  Simplifies development and gives faster time to market.
- Standardized APIs.
  Ensures portability between implementations from different vendors.
- Simplifies connectivity.
  Provides, for instance, JMS, CORBA, RMI and J2EE Connectors support.
- Uses annotations to avoid separation between code and metadata.
  Also helps avoiding XML configuration files.

Services and APIs of the Java EE Platform

The following are some examples of services provided by the Java EE platform:

- Resource management.
- Lifecycle management (EJB and servlets).
- Distributed communication (EJB).
- Threading (EJB).
- Scaling (EJB).
- Transaction management (EJB).
- Infrastructure for components (servlets).
- Infrastructure for communication (servlets).
- Infrastructure for session management (servlets).

The Java EE platform provides the following APIs related to web services. Many of these APIs are included in Java SE 6.

- JAX-WS
- JAX-RPC
- JAXB
- SAAJ
- JAXP
  JAXP consists of the following APIs:
  - SAX
  - DOM
  - StAX
  - An XSLT API, originally developed under the name TrAX.
- JAXR

7.2 Benefits

Explain the benefits of using the Java EE platform for creating and deploying web service applications.

Some benefits of using the Java EE platform for creating and deploying web services are:

- Less code needs to be written.
  Mainly because the amount of boilerplate code has been reduced.
  This gives the benefit of, among other things, reduced development time.
- Easy to learn and use.
- Supports the latest standards and programming styles.
  For instance, full support for W3C XML, support for WS-I Basic Profile 1.1, WSDL 1.1 etc.
- Provides complete support for developing and deploying web service applications:
  From the NetBeans IDE and other development tools, all the different APIs that might be needed and, finally, to the GlassFish application server, on which the web service applications can be deployed.
- Allows for interoperability between Java web service servers and/or clients and web services and/or clients developed on other platforms.

7.3 Functions and Capabilities

Describe the functions and capabilities of the JAXP, DOM, SAX, StAX, JAXR, JAXB, JAX-WS and SAAJ in the Java EE platform.

References:
http://java.sun.com/javaee/overview/faq/javatech_xml.jsp

API	Functions	Capabilities
JAXP	Parse and transform XML documents.	Validating and parsing XML documents. Perform data and structural transformation of XML documents.
DOM	Read and write XML data.	Parses an entire XML document and maintains an in-memory representation of the document.
SAX	Read XML data.	Push-parser which, using callbacks, sends information about the XML document being read to client having implemented certain callback interfaces.
StAX	Read and write XML data.	Pull-parser from which clients can retrieve information about the XML document being read as needed.
JAXR	Provides an uniform, standard, Java API for accessing XML registries.	Query, insert, remove and modify data in XML registries.
JAXB	Marshalling and unmarshallaing of XML data to/from Java objects.	Validate XML data. From XML schema, generate Java interfaces and classes that represents the schema. Access data of unmarshalled XML document in any order.
JAX-WS	Develop SOAP and RESTful web services and web service clients.	WSDL 1.1 mapping to Java. Java to WSDL 1.1 mapping. Web service client runtime. Web service runtime. Uses Java annotations. Message-processing plug-in framework (handlers). SOAP & HTTP binding.
SAAJ	Produce and consume SOAP 1.1 and 1.2 messages and SOAP messages with attachments.	Create, manipulate and send SOAP 1.1 and 1.2 messages, with or without attachments.

7.4 Role of the WS-I Basic Profile

Describe the role of the WS-I Basic Profile when designing Java EE Web services.

References:
http://www.ws-i.org/Profiles/BasicProfile-1.1.html

The role of the WS-I Basic Profile is to:
Provide interoperability guidance for core web service specifications, such as SOAP, WSDL and UDDI.
From the WS-I Basic Profile itself:
"...the WS-I Basic Profile 1.1, consisting of a set of non-proprietary Web services specifications, along with clarifications, refinements, interpretations and amplifications of those specifications which promote interoperability."

Even though applied to its fullest extent, the WS-I Basic Profile does not guarantee interoperability, but merely increases the chances that interoperability is achieved.

8. Security

8.1 Security Mechanisms

Explain basic security mechanisms including: transport level security, such as basic and mutual authentication and SSL, message level security, XML Encryption, XML Digital Signature, and federated identity and trust.

References:
http://en.wikipedia.org/wiki/Basic_access_authentication
http://en.wikipedia.org/wiki/Transport_Layer_Security
http://www.w3.org/TR/xmldsig-core/
http://en.wikipedia.org/wiki/XML_Digital_Signature
http://en.wikipedia.org/wiki/XML_Encryption
http://www.w3.org/TR/xmlenc-core/
http://en.wikipedia.org/wiki/Federated_identity

Transport Level Security

Transport level security refers to securing the connection between a web service and a client application using one or both of the following mechanisms:

- HTTP basic authentication.
- Secure Socket Layer (SSL).

HTTP Basic Authentication

HTTP Basic Authentication between a client and a web service is performed in the following steps:

- The client sends a user name and a password.
- The web service authenticates the client and returns an authentication header.
- The client includes the authentication header in all subsequent requests.

HTTP Basic Authentication in itself only provides authentication of a user, but neither confidentiality nor data integrity.

Secure Socket Layer

Secure Socket Layer provides confidentiality and integrity by encrypting all the data sent between a client and a web service at the transport layer.
SSL provides two modes of authentication; unilateral authentication and mutual authentication.

With unilateral authentication the client verifies the web service's certificate with a third party, such as a Certification Authority. Note that the client is still anonymous to the web service.
With mutual authentication both the client and the web service verifies each other's certificates with a third party.

There are some drawbacks with SSL security:

- Does not allow intermediaries access to SOAP messages.
  All of the communication between the server is encrypted, which makes it impossible for intermediaries to process any messages.
- “All-or-nothing” - either entire messages sent between client and web service are encrypted or nothing.
- Once a message is decrypted, as it is received, security is completely removed.
- SSL does not really work with other transport protocols.

Message Level Security

Message level security refers to applying one, or both, of the following to a message sent between a web service and a client:

- Digitally signing the message.
- Encrypting whole or part of the message.

Additionally, if the web service and client exchange multiple messages, a shared security context can be specified.
Message level security can aid in assuring the following:

- Confidentiality.
- Integrity.
- Authenticity.

There are two W3C recommendations that addresses message level security; XML Signature and XML Encryption.

(continued on next page)

XML Signature

The purpose of a signature is to identify what is signed and make it impossible to alter the signature or what has been signed without detection.
Digitally signing an XML document provide:

- A way to verify the integrity of the message.
- Message authentication.
  Ensure that the message has not been tampered with, nor have the checksum verifying the integrity of the message.
- A way to verify the identity of message sender.

A signature of an XML element in an XML document is created by:

- Transforming the XML data to be signed to a standardized form (canonicalization).
- Calculating a digest of the canonicalized XML data.

The following shows an example of a signature adhering to the XML Signature standard:

```
<Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
        <CanonicalizationMethod Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
        <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
        <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
            <Transforms>
                <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
            </Transforms>
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK.../DigestValue>
        </Reference>
    </SignedInfo>
    <SignatureValue>...</SignatureValue>
    <KeyInfo>
        <KeyValue>
            <DSAKeyValue>
                <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
            </DSAKeyValue>
        </KeyValue>
    </KeyInfo>
</Signature>
```

The example is taken from the XML Signature specification (Copyright 2008 The Internet Society & W3C).
XML Signature can be used to sign any kinds of resources that can be accessed via an URL. It is typically used to sign XML documents. There are three kinds of signatures:

Signature Type	Description
Detached Signature	The signature signs data that is located outside of the XML document in which the signature occurs.
Enveloped Signature	The signature signs part of the XML document in which the signature occurs.
Enveloping Signature	The signature signs data contained in the signature itself.

XML Encryption

XML Encryption enables encryption of arbitrary data, XML documents XML elements or XML element contents with the result being an XML document which contains, or references, the encrypted data.
Consider this example, taken from the XML Encryption specification (Copyright 2002 W3C) . First the unencrypted XML document:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <CreditCard Limit='5,000' Currency='USD'>
      <Number>4019 2445 0277 5567</Number>
      <Issuer>Example Bank</Issuer>
      <Expiration>04/02</Expiration>
    </CreditCard>
</PaymentInfo>
```

In the first example of encryption, we'll look at the above XML document with the <CreditCard> element encrypted using XML Encryption:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
     xmlns='http://www.w3.org/2001/04/xmlenc#'>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
</PaymentInfo>
```

This second example of encryption shows how the content of the <CreditCard> element has been encrypted:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <CreditCard Limit='5,000' Currency='USD'>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
       Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData>
    </CreditCard>
</PaymentInfo>
```

The final example of encryption shows how the credit card number, originally contained in the <Number> element, has been encrypted.

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
    <Name>John Smith</Name>
    <CreditCard Limit='5,000' Currency='USD'>
      <Number>
        <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
         Type='http://www.w3.org/2001/04/xmlenc#Content'>
          <CipherData>
            <CipherValue>A23B45C56</CipherValue>
          </CipherData>
        </EncryptedData>
      </Number>
      <Issuer>Example Bank</Issuer>
      <Expiration>04/02</Expiration>
    </CreditCard>
</PaymentInfo>
```

Federated Identity and Trust

Identity federation means the linking of a person's user information across otherwise separate boundaries, such as software systems, without centrally storing information.
Federation is enabled through the use of open standards and openly published specifications, enabling interoperability for common use-cases between multiple parties.
An example of identity federation is single sign-on, which enables a user of a multiple systems to log in once and gains access to all the systems.

Example
A person is a customer of a travel agency and has an account in their system. The travel agency uses a federated identity system cooperating with a chain of hotels. The travel agency's system and the hotel chain's system is in a circle of trust (see below). This enables the customer to log into the travel agency's system and book a journey, then continuing on to the hotel chain's system to book a room. The identity is carried over from the travel agency's system, enabling the user to book the room without having to log in to the hotel chain's system.
The hotel chain's system trusts the authentication credentials issued by the travel agency's system.

A circle of trust is described as a group in which each participant is trusted to describe:

- The process used to identify a user.
- The authentication system used.
- Policies related to the handling of authentication credentials.

Each of the participants in a circle of trust can examine the descriptions of other participants and decide whether to trust these or not.

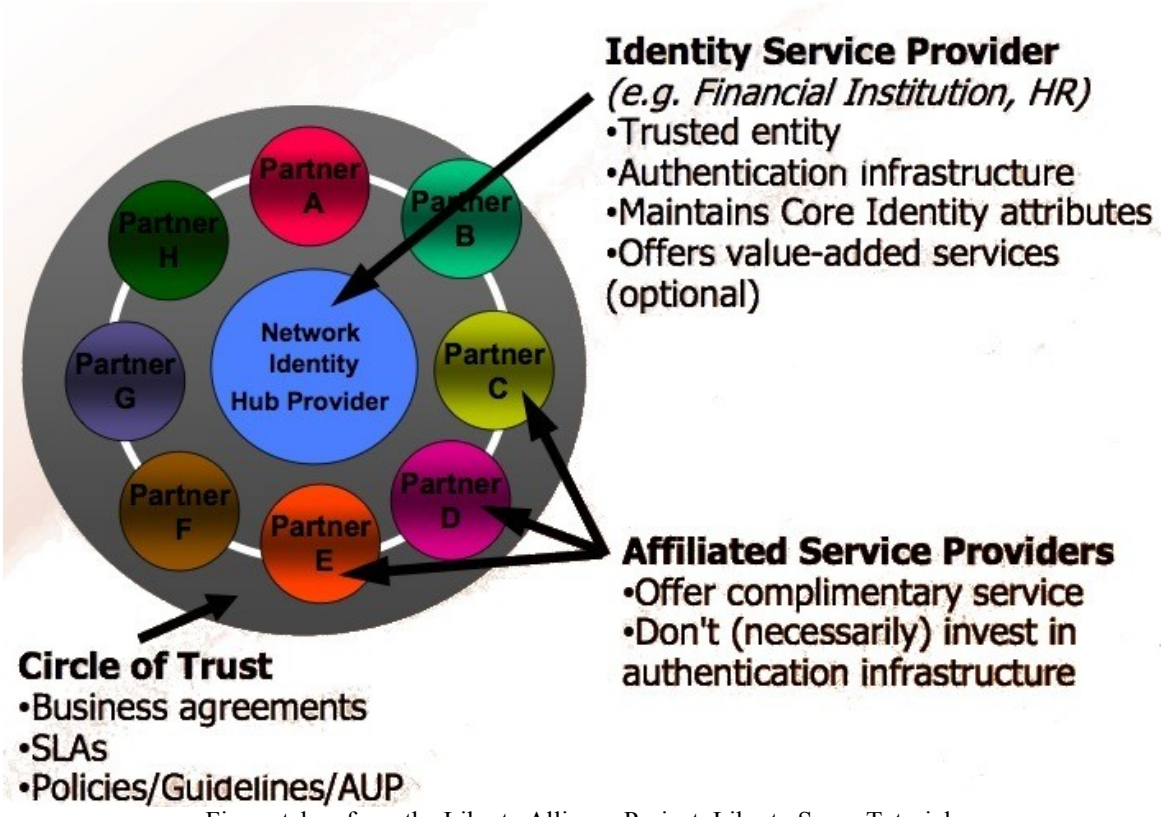


Figure taken from the Liberty Alliance Project, Liberty Specs Tutorial.

8.2 Web Services Security Initiatives and Standards

Identify the purpose and benefits of Web services security oriented initiatives and standards such as Username Token Profile, SAML, XACML, XKMS, WS-Security, and the Liberty Project.

References:
http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf
http://en.wikipedia.org/wiki/Saml
http://www.xml.com/pub/a/2005/01/12/saml2.html
http://sunxacml.sourceforge.net/guide.html
http://en.wikipedia.org/wiki/XKMS
http://www.w3.org/TR/xkms2/
http://www.verisign.com/developer/xml/xkms.html
http://www.ibm.com/developerworks/webservices/library/ws-best11/
http://www.projectliberty.org/liberty/resource_center/tutorials

It is highly recommended to consult the above references for a deeper understanding of the different standards and initiatives.

Standard/Initiative	Purpose	Benefits
Username Token Profile	A UsernameToken is a means by which a web service consumer can supply authentication information, such as a username and a password, shared secret or password equivalent, to authenticate an identity to a web service producer.	Standardization. Interoperability. Increased security (if recommendations are followed),
SAML	Defines an XML framework (the Security Assertion Markup Language) for exchanging authentication, authorization and attribute information. Makes single sign-on, distributed transactions and authorization services possible.	Platform neutrality. Loose coupling. Improved experience for clients using services in different domains.
XACML	A policy language and an access control decision request/response language. The policy language is used to describe general access control requirements and the request/response language is used to query whether or not a given action should be allowed, and interpret the result.	Standardization. Interoperability. Code reusability. Extensibility. Flexibility. Can be used in any environment. Enables distribution of policies to multiple locations.

(continued on next page)

Standard/Initiative	Purpose	Benefits
XKMS	Delegate authentication, digital signature, and encryption services, such as certificate processing and revocation status checking, to an external XKMS service.	Reduce complexity and workload of applications requiring the mentioned services and PKI (public key infrastructure). Easy to use. Quick to develop. XKMS is an open standard. Isolates applications using XKMS from the technology behind XKMS servers.
WS-Security	Enable SOAP message content integrity, confidentiality and SOAP message authentication. Provide an extensible framework that can be used to implement various security mechanisms. Provide end-to-end message security and not just transport level security.	Flexibility. Extensibility. Standardization. Interoperability.
Liberty Project	Establish open standards, best practices and guidelines for federated identity management. Provide an open and secure standard for SSO. Promote interoperability.	Improved user experience. Increased security and privacy. Interoperability.

8.3 JavaEE Based Web Service Security

Given a scenario, implement Java EE based web service web-tier and/or EJB-tier basic security mechanisms, such as mutual authentication, SSL, and access control.

References:
http://www.java-tips.org/java-ee-tips/java-api-for-xml-web-services/using-jax-ws-based-web-services-wit.html

Before being able to start coding the examples in this section, we need to prepare the keys used for authentication, decryption and encryption. Since this is a non-trivial process, it will be described in detail in the following section.

Setting Up for Mutual Authentication

In order to be able to use mutual authentication, we need to do the following:

- Create a client keystore.
- Export the client certificate from the client keystore.
- Import the client certificate to the server truststore.
- Create a client truststore.
- Export the server certificate from the server keystore.
- Import the server certificate to the client truststore.

The examples in this section will use the GlassFish application server in its default configuration. The GlassFish installation directory will be referred to as GLASSFISH_DIR. We also need to work from some terminal window, since we need to use the *keytool* command to create and manipulate key and truststores.

- Create a directory named “client” at some arbitrary location.
- Create a directory named “server” inside the “client” directory created in the previous step.
- Back up the cacerts.jks file located in the GLASSFISH_DIR/domains/domain1/config/ directory.
- Copy the cacerts.jks file that you just backed up to the “server” directory.
- Back up the keystore.jks file located in the GLASSFISH_DIR/domains/domain1/config/ directory.
- Copy the keystore.jks file that you just backed up to the “server” directory.
- Open a terminal window and go to the “client” directory.
- Using the keytool command, create the client keystore and key as follows. The questions ansked by the keytool program can be answered more or less arbitrarily, as long as you remember the alias of the key you generate.

```
keytool -genkey -alias client -keypass changeit -storepass changeit -keystore
client_keystore.jks
What is your first and last name?
  [Unknown]:  My Client
What is the name of your organizational unit?
  [Unknown]:  Client Org Unit
What is the name of your organization?
  [Unknown]:  Client Org
What is the name of your City or Locality?
  [Unknown]:  Client City
What is the name of your State or Province?
  [Unknown]:  Client State
What is the two-letter country code for this unit?
  [Unknown]:  US
```

```
Is CN=My Client, OU=Client Org Unit, O=Client Org, L=Client City, ST=Client State, C=US
correct?
  [no]:  yes
```

- Using the keytool command, export the client certificate.

```
keytool -export -alias client -keystore client_keystore.jks -storepass changeit -file
client.cer
Certificate stored in file <client.cer>
```

- Import the client certificate to the server truststore.

```
keytool -import -v -trustcacerts -alias client -keystore server/cacerts.jks -keypass
changeit -file client.cer
Enter keystore password:  changeit
Owner: CN=My Client, OU=Client Org Unit, O=Client Org, L=Client City, ST=Client State,
C=US
Issuer: CN=My Client, OU=Client Org Unit, O=Client Org, L=Client City, ST=Client State,
C=US
Serial number: 49cadb4c
Valid from: Thu Mar 26 19:33:00 CST 2009 until: Wed Jun 24 19:33:00 CST 2009
Certificate fingerprints:
       MD5:  00:E7:89:7B:FA:2D:65:21:CC:65:A8:AA:FF:FC:F7:2F
       SHA1: 23:9F:2D:E1:6B:27:ED:CA:A3:83:90:47:9C:CF:55:C7:59:A6:14:2C
Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing server/cacerts.jks]
```

- Change directory to the “server” directory.
- Export the server certificate.

```
keytool -export -alias s1as -keystore keystore.jks -storepass changeit -file server.cer
Certificate stored in file <server.cer>
```

- Go back to the “client” directory.
- In one single step, create the client truststore and import the server certificate into it.

```
keytool -import -v -trustcacerts -alias s1as -keystore client_cacerts.jks -storepass
changeit -keypass changeit -file server/server.cer
Owner: CN=Bo5b.local, OU=Sun Java System Application Server, O=Sun Microsystems, L=Santa
Clara, ST=California, C=US
Issuer: CN=Bo5b.local, OU=Sun Java System Application Server, O=Sun Microsystems,
L=Santa Clara, ST=California, C=US
Serial number: 47e86844
Valid from: Tue Mar 25 19:49:40 CST 2008 until: Fri Mar 23 19:49:40 CST 2018
Certificate fingerprints:
       MD5:  9A:70:E1:43:87:71:6B:A1:52:E0:B7:55:93:B6:DA:74
       SHA1: D8:04:53:C4:46:88:8B:30:EA:DC:1C:1C:1A:90:2B:ED:49:5C:9F:27
Trust this certificate? [no]:  yes
Certificate was added to keystore
[Storing client_cacerts.jks]
```

- Copy the files cacerts.jks and keystore.jks from the “server” directory to the GLASSFISH_DIR/domains/domain1/config/ directory, replacing any files with the same names.
- If GlassFish is running, restart it, so that the modified keystore and truststore takes effect.
- Save the client_cacerts.jks and client_keystore.jks files – the web service clients in the examples below will need these files.

Web Tier Web Services

In this section we will implement a servlet-based web service and associated client and then show how to apply access control, SSL and mutual authentication to this web service and its client.

Servlet Based Web Service

First we will implement the basic servlet-based web service without any encryption:

- Create a dynamic web project in Eclipse that uses the GlassFish server.
  I have called my project JAX-WS_GreetingServletMutualAuth.
- Implement the web service as follows:

```
package com.ivan;

import java.util.Date;
import javax.jws.WebService;

/**
 * This class implements the JAX-WS Hello World web service.
 * This is the servlet version of the web service.
 *
 * @author Ivan A Krizsan
 */
@WebService
public class HelloWorldWS
{
    /* Constant(s): */

    /* Instance variable(s): */
    @Resource
    private WebServiceContext mWSContext;

    /**
     * Greets the user by composing a greeting-string including supplied
     * message.
     *
     * @param inMessage Message to be included in greeting string.
     * @return Greeting string.
     */
    public String hello(final String inMessage)
    {
        return "Hello from Servlet WS: " + inMessage + " " +
            new Date() + ", the principal is: " +
            mWSContext.getUserPrincipal();
    }
}
```

- If not already present, create a web.xml file in the WEB-INF directory with the following contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
    <display-name>JAX-WS_GreetingServletMutualAuth</display-name>

</web-app>
```

- If not already present, create a sun-web.xml file in the WEB-INF directory with the following contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/JAX-WS_GreetingServletMutualAuth</context-root>


    <!-- The stuff below is just default settings. -->
    <class-loader delegate="true" />
    <jsp-config>
        <property name="keepgenerated" value="true"/>
    </jsp-config>

</sun-web-app>
```

- Start GlassFish.
- Deploy the web service to GlassFish.

Web Service Client

The web service client used in this example is a standalone Java program that only relies on the Java 6 runtime environment. To set up an Eclipse project for the client:

- Create a Java project.
- Copy the client keystore and truststore files, client_cacerts.jks and client_truststore.jks, into the root of the project.
- Implement the main client class:
  Note that there will be errors in this class until we have generated additional classes needed on the client side in the next step.

```
package com.ivan.client;

import javax.net.ssl.HostnameVerifier;
import javax.net.ssl.HttpsURLConnection;
import javax.net.ssl.SSLSession;
import com.ivan.HelloWorldWS;
import com.ivan.HelloWorldWSService;

/**
 * Standalone static JAX-WS client invoking the secured Hello service.
 * When launching the program, the following VM flags need to be set
 * for SSL:
 * -Djavax.net.ssl.trustStore=client_cacerts.jks
 * -Djavax.net.ssl.trustStorePassword=changeit
 *
 * Additionally, for mutual authentication:
 * -Djavax.net.ssl.keyStore=client_keystore.jks
 * -Djavax.net.ssl.keyStorePassword=changeit
 *
 * Finally, when debugging SSL handshakes etc:
 * -Djavax.net.debug=all
 *
 * @author Ivan A Krizsan
 */
public class HelloWSClient
{

    static
    {
        /*
         * Java by default verifies that the certificate CN (Common Name) is
```

```
         * the same as host name in the URL. If the CN in the certificate is
         * not the same as the host name, your web service client fails.
         * This piece of code allows for using localhost as host name
         * with a certificate in which the CN does not match.
         * This is meant to be a workaround while developing the web
         * service and clients and SHOULD be removed in the production
         * version.
         */
        HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier()
        {
            public boolean verify(String hostname, SSLSession session)
            {
                if (hostname.equals("localhost"))
                {
                    return true;
                }
                return false;
            }

        });
    }

    /* Instance variable(s): */
    private HelloWorldWSService mHelloWorldService;

    public static void main(String[] args)
    {
        HelloWSClient theClient = new HelloWSClient();
        theClient.callService();
    }

    private void callService()
    {
        mHelloWorldService = new HelloWorldWSService();

        System.out.println("Service object: " + mHelloWorldService);
        HelloWorldWS thePort = mHelloWorldService.getHelloWorldWSPort();

        String theResponse = thePort.hello("Steven Segal");

        System.out.println("Response from web service: " + theResponse);
    }
}
```

- Create the Ant script used to generate the client artifacts.
  It is to be located in the root of the project. Sections marked with red may need modifications, depending on your environment.
  Note! Do not change the WSDL location to HTTPS, it is not needed!

```
<?xml version="1.0"?>

<project default="main" basedir=".">

    <property name="class-dir" value="${basedir}/bin" />
    <property name="wsimport-outdir" value="${basedir}/wsimport_generated" />
    <property name="gen-classdir" value="${wsimport-outdir}/com/" />
    <property name="src-outdir" value="${basedir}/src/" />
    <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport" />
    <property name="wsdl-location" value="http://localhost:8080/JAX-
WS_GreetingServletMutualAuth/HelloWorldWSService?wsdl" />

    <target name="main">
        <exec executable="${wsimport-cmd}">
            <arg value="-verbose" />

            <!-- Specify where to write other generated files (i.e. class files). -->
            <arg value="-d" />
            <arg value="${wsimport-outdir}" />

            <!-- Specify where to write generated source files. -->
            <arg value="-s" />
            <arg value="${src-outdir}" />

            <!-- Keep generated source files. -->
            <arg value="-keep" />
```

```
            <!-- Specify location of WSDL from which to generate the artifacts. -->
            <arg value="${wsdl-location}" />
        </exec>

        <!--
            Delete the directory containing the generated classes and its
            contents.
        -->
        <delete dir="${gen-classdir}" />
    </target>
</project>
```

- Create a folder named “wsimport_generated” in the root of the project.
- Run the above Ant script.
  The following source-code files will be created in the project: Hello.java, HelloResponse.java, HelloWorldWS.java, HelloWorldWSService.java, ObjectFactory.java and package-info.java.
  Remember that you may need to refresh the project in Eclipse to see the files!

With the web service deployed and the client ready, you should now be able to run the client and receive a response from the web service without any errors occurring.
The console output should be along the following lines:

```
Service object: com.ivan.HelloWorldWSService@6f03de90
Response from web service: Hello from Servlet WS: Steven Segal Thu Mar 26 21:20:27 CST
2009, the principal is: null
```

Now that we have a working web service and a client, we can start adding different kinds of security. First out is basic access control.

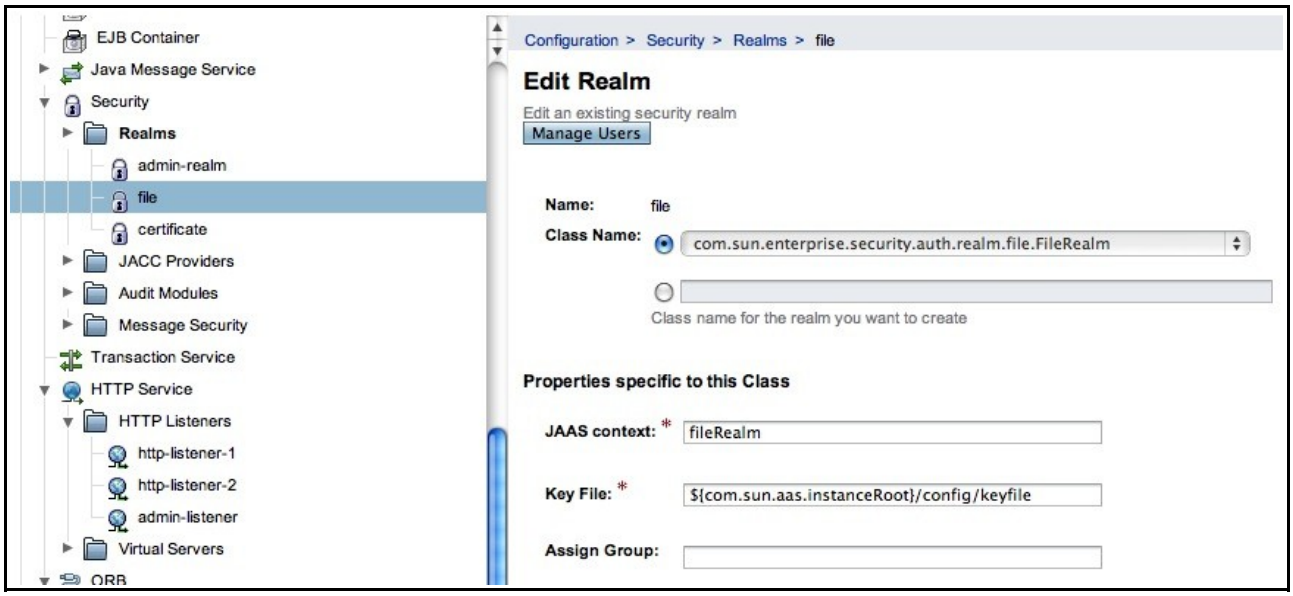
Access Control

Access control means that we want to control who can access our web service. Typically, this means requiring clients to provide a name and a password, something that can be done by adding basic authentication to our web service.

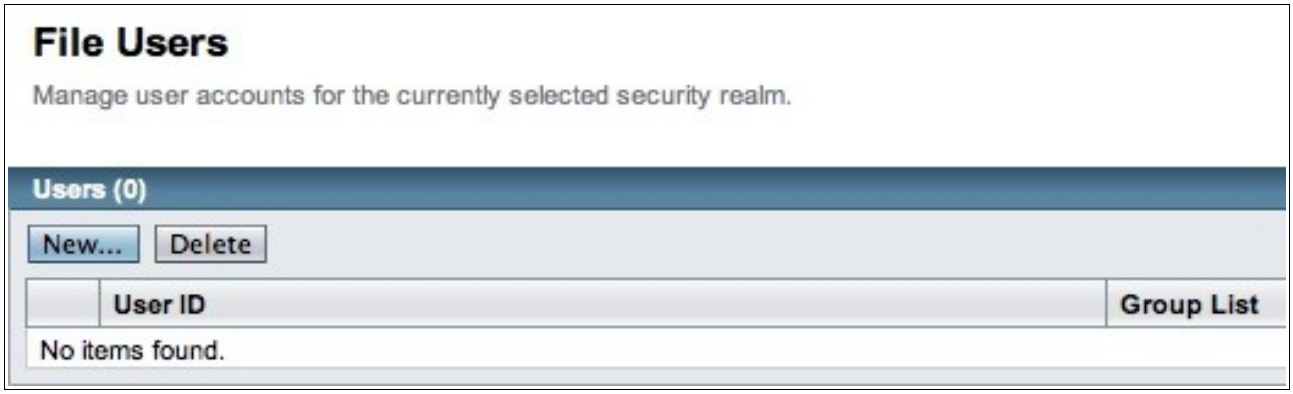
Creating A User in GlassFish

First of all, we need to create a user in GlassFish:

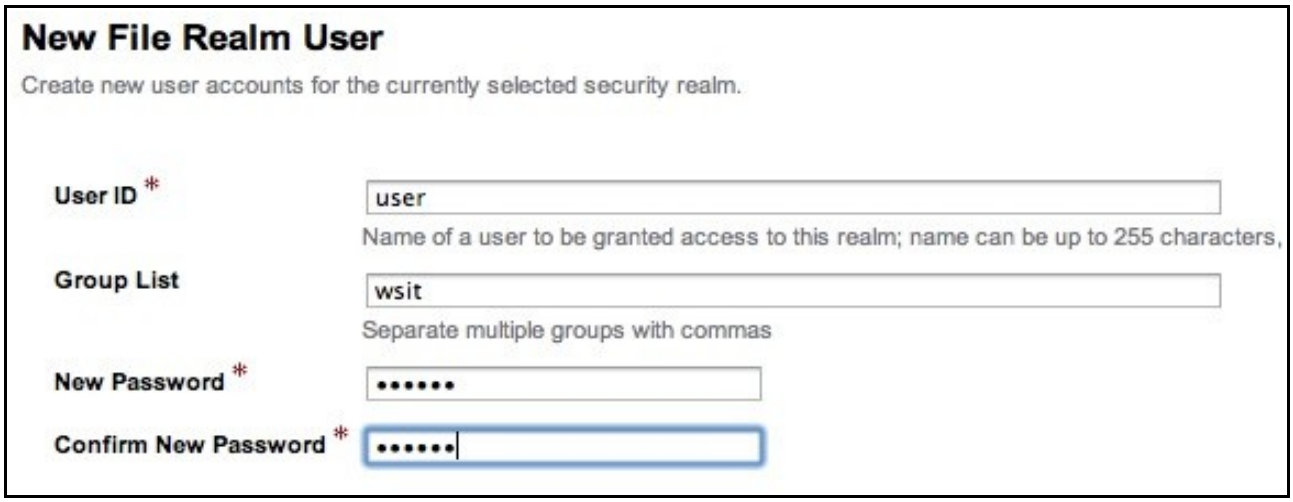
- In the GlassFish administration console, go to the Security → Realms → file.



- Click the Manage Users button.
  The file realm user list will appear, probably empty if you haven't already added users.



- Click the New... button above the user list and fill in user data as below.
  I used the password “secret” for this example.



- When finished, click the OK button in the upper right corner.
  GlassFish does not need to be restarted!

Configuring the Server

On the server side, we only need to modify the two deployment descriptors.
First, this is what the new web.xml deployment descriptor will look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
    <display-name>JAX-WS_GreetingServletMutualAuth</display-name>

    <security-constraint>
        <display-name>SecurityConstraint</display-name>
        <web-resource-collection>
            <web-resource-name>Secure Area</web-resource-name>
            <url-pattern>/HelloWorldWSService</url-pattern>

            <!--
                JSR-109 specifies that http-method POST must be used.
                If we include GET here, access to the WSDL will also
                be protected, otherwise not.
             -->
            <http-method>POST</http-method>
        </web-resource-collection>

        <!-- Here we specify which roles can access the web service. -->
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>
    </security-constraint>

    <!--
        Access control using name and password require the auth-method
        to be BASIC.
    -->
    <login-config>
        <auth-method>BASIC</auth-method>
```

```
    </login-config>

    <!-- Declare the security role used above. -->
    <security-role>
        <role-name>user</role-name>
    </security-role>
</web-app>
```

The sun-web.xml deployment descriptor will look like this when configured for basic authentication:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/JAX-WS_GreetingServletMutualAuth</context-root>

    <!-- Mapping of the user allowed to access the web service. -->
    <security-role-mapping>
        <role-name>user</role-name>
        <group-name>wsit</group-name>
    </security-role-mapping>

    <!-- The stuff below is just default settings. -->
    <class-loader delegate="true" />
    <jsp-config>
        <property name="keepgenerated" value="true"/>
    </jsp-config>
</sun-web-app>
```

Remember to deploy the updated web service to GlassFish after having finished the modifications!

If you now run the web service client, you should receive an error saying “request requires HTTP authentication: Unauthorized”. This is perfectly normal, since the client did not supply a user name and a password.

Modifying the Client

In order to be able to use the now protected web service, the client is required to enclose authorization credentials, that is a user name and a password.
In the *HelloWSClient* class in the client code, modify the *callService()* method. The modified version should look like this:

```
    ...
    private void callService()
    {
        mHelloWorldService = new HelloWorldWSService();

        System.out.println("Service object: " + mHelloWorldService);
        HelloWorldWS thePort = mHelloWorldService.getHelloWorldWSPort();

        /* Set user name and password for basic authentication. */
        BindingProvider theBindingProvider = (BindingProvider)thePort;
        Map<String, Object> theRequestContext = theBindingProvider.getRequestContext();
        theRequestContext.put("javax.xml.ws.security.auth.username", "user");
        theRequestContext.put("javax.xml.ws.security.auth.password", "secret");

        String theResponse = thePort.hello("Steven Segal");

        System.out.println("Response from web service: " + theResponse);
    }
    ...
```

Having applied these modifications, the client should now run without errors.

SSL

When applying the modifications to add transport-layer security to the web service, please start from the web service and client that does not have any security added, as described above.

Configuring the Server

To configure the servlet-based web service to use SSL encryption, perform the following modifications:

- Modify the web.xml deployment descriptor to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
    <display-name>JAX-WS_GreetingServletMutualAuth</display-name>

    <security-constraint>
        <display-name>SecurityConstraint</display-name>
        <web-resource-collection>
            <web-resource-name>Secure Area</web-resource-name>
            <url-pattern>/HelloWorldWSService</url-pattern>

            <!--
                JSR-109 specifies that http-method POST must be used.
                If we include GET here, access to the WSDL will also
                be protected, otherwise not.
             -->
            <http-method>POST</http-method>
        </web-resource-collection>

        <!--
            Plain SSL and mutual authentication both require CONFIDENTIAL
            transport-guarantee to be configured.
        -->
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
</web-app>
```

- Redeploy the web service to GlassFish.

Modifying the Client

The client requires two changes; modify the URL of the web service to use HTTPS and add JVM parameters to the launch configuration in order for the client to be able to use the client truststore we created earlier.

- Modify the service URL in the generated class *HelloWorldWSService* to use HTTPS (marked with red).

```
@WebServiceClient(name = "HelloWorldWSService", targetNamespace = "http://ivan.com/",
wsdlLocation = "http://localhost:8080/JAX-
WS_GreetingServletMutualAuth/HelloWorldWSService?wsdl")
public class HelloWorldWSService
    extends Service
{

    private final static URL HELLOWORLDWSSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("https://localhost:8181/JAX-
WS_GreetingServletMutualAuth/HelloWorldWSService?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDWSSERVICE_WSDL_LOCATION = url;
    }
```

- Add the following VM arguments to the launch configuration of the client:
  -Djavax.net.ssl.trustStore=client_cacerts.jks
  -Djavax.net.ssl.trustStorePassword=changeit
  -Djavax.net.debug=all

With these changes in place and after having deployed the modified web service, the client will produce a result similar to what we already have seen. Additionally there will be quite some logging output showing the SSL handshaking between the server and client.

Mutual Authentication

When applying the modifications to add mutual authentication to the web service, please start from the web service and client that does not have any security added, as described above.

Configuring the Server

To configure the servlet-based web service to use mutual authentication, perform the following modifications:

- Modify the web.xml deployment descriptor to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">
    <display-name>JAX-WS_GreetingServletMutualAuth</display-name>

    <security-constraint>
        <display-name>SecurityConstraint</display-name>
        <web-resource-collection>
            <web-resource-name>Secure Area</web-resource-name>
            <url-pattern>/HelloWorldWSService</url-pattern>

            <!--
                JSR-109 specifies that http-method POST must be used.
                If we include GET here, access to the WSDL will also
                be protected, otherwise not.
             -->
            <http-method>POST</http-method>
        </web-resource-collection>

        <!--
            For mutual authentication, this element must be included
            with a role-name that is mapped to the client certificate,
            see the sun-web.xml deployment descriptor!
        -->
        <auth-constraint>
            <role-name>user</role-name>
        </auth-constraint>

        <!--
            Plain SSL and mutual authentication both require CONFIDENTIAL
            transport-guarantee to be configured.
        -->
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>

    <!--
        Mutual authentication require the auth-method to be CLIENT-CERT.
    -->
    <login-config>
        <auth-method>CLIENT-CERT</auth-method>
    </login-config>

    <!-- Declare the security role used above. -->
    <security-role>
        <role-name>user</role-name>
    </security-role>
</web-app>
```

- Modify the sun-web.xml deployment descriptor to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/JAX-WS_GreetingServletMutualAuth</context-root>

    <!--
        In order for GlassFish to realize the connection between the
        security role used in the web.xml deployment descriptor and the
        client certificated, we need to map the role name to the
        certificate principal name, which can be obtained either when
        creating the certificate or by using the keytool command to list
        the contents of the client keystore.
    -->
    <security-role-mapping>
        <role-name>user</role-name>
        <principal-name>CN=My Client, OU=Client Org Unit, O=Client Org, L=Client City,
ST=Client State, C=US</principal-name>
    </security-role-mapping>

    <!-- The stuff below is just default settings. -->
    <class-loader delegate="true" />
    <jsp-config>
        <property name="keepgenerated" value="true"/>
    </jsp-config>
</sun-web-app>
```

Modifying the Client

The client requires two changes; modify the URL of the web service to use HTTPS and add JVM parameters to the launch configuration in order for the client to be able to use the client truststore we created earlier.

- Modify the service URL in the generated class *HelloWorldWSService* to use HTTPS (marked with red). Note that the WSDL URL should not be modified!

```
@WebServiceClient(name = "HelloWorldWSService", targetNamespace = "http://ivan.com/",
wsdlLocation = "http://localhost:8080/JAX-
WS_GreetingServletMutualAuth/HelloWorldWSService?wsdl")
public class HelloWorldWSService
    extends Service
{

    private final static URL HELLOWORLDWSSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new URL("https://localhost:8181/JAX-
WS_GreetingServletMutualAuth/HelloWorldWSService?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDWSSERVICE_WSDL_LOCATION = url;
    }
```

- Add the following VM arguments to the launch configuration of the client:
  -Djavax.net.ssl.trustStore=client_cacerts.jks
  -Djavax.net.ssl.trustStorePassword=changeit
  -Djavax.net.ssl.keyStore=client_keystore.jks
  -Djavax.net.ssl.keyStorePassword=changeit
  -Djavax.net.debug=all

With these changes in place and after having deployed the modified web service, the client will produce the a result similar to what we already have seen. Again, there will be quite some logging output showing the SSL handshaking between the server and client.
To verify that mutual authentication really has taken place, search for the string “CertificateRequest” in the SSH log.

EJB Tier Web Services

In this section we will implement a EJB-based web service and associated client and then show how to apply access control, SSL and mutual authentication to this web service and its client.

Note that GlassFish must be configured as described in the section Setting Up for Mutual Authentication above, in order for the SSL and mutual authentication examples below to execute properly!

EJB Based Web Service

First we will implement the basic EJB-based web service without any encryption:

- Create an EJB project in Eclipse that uses the GlassFish server.
  I have called my project JAX-WS_GreetingEJBMutualAuth.
- Implement the web service as follows:

```
package com.ivan;

import java.util.Date;
import javax.ejb.Stateless;
import javax.jws.WebService;

/**
 * This class implements an EJB greeting web service that is to be
 * protected by mutual authentication.
 *
 * @author Ivan A Krizsan
 */
@Stateless(name="HelloWorldEJBWS")
@WebService()
public class HelloWorldEJBWS
{
    /**
     * Greets the user by composing a greeting-string including supplied
     * message.
     *
     * @param inMessage Message to be included in greeting string.
     * @return Greeting string.
     */
    public String hello(final String inMessage)
    {
        return "Hello from EJB WS " + inMessage + " " + new Date();
    }
}
```

- If not already present, create a sun-web.xml file in the META-INF directory with the following contents.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB
3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
      <enterprise-beans>
      </enterprise-beans>
</sun-ejb-jar>
```

- Start GlassFish.
- Deploy the web service to GlassFish.

Web Service Client

The client of the EJB web service is implemented in a manner similar to that of the servlet-based web service client. Due to this similarity, the implementation is left as an exercise for the reader. Finally, when running the client, the EJB-based web service should respond with a message similar to that of the servlet-based web service client.

Access Control

EJB-based web service endpoints can also be configured for access control using a user name and a password. Before configuring the server, make sure that you have created a user in GlassFish, as described in the section on Access Control for the servlet-based web service.

Configuring the Server

On the server-side, only the sun-ejb.xml deployment descriptor needs to be modified. With modifications in place, it should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB
3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>HelloWorldEJBWS</ejb-name>
            <webservice-endpoint>
                <port-component-name>HelloWorldEJBWS</port-component-name>

                <!--
                    For access control, this element must be
                    configured to use the BASIC authentication
                    method.
                    Also note that a user must be properly configured
                    for the file-realm in GlassFish.
                -->
                <login-config>
                    <auth-method>BASIC</auth-method>
                </login-config>
            </webservice-endpoint>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

Remember to deploy the updated web service to GlassFish after having finished the modifications!

If you now run the web service client, you should receive an error saying “request requires HTTP authentication: Unauthorized”.

Modifying the Client

In order to be able to use the now protected web service, the client is required to enclose credentials; a user name and a password. Adding user name and password to the client is done in exactly the same way as with the servlet-based web service, please refer to the Access Control section for the servlet-based web service above.

Having applied the modification, the client should now run without errors.

SSL

When applying the modifications to add transport-layer security to the web service, please start from the web service and client that does not have any security added, as described above. Adding SSL to the EJB-based web service, will not only cause the access to the actual web service be encrypted, but the access to the WSDL as well.

Configuring the Server

To configure the EJB-based web service to use SSL encryption, perform the following modifications:

- Modify the sun-ejb.xml deployment descriptor to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB
3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>HelloWorldEJBWS</ejb-name>
            <webservice-endpoint>
                <port-component-name>HelloWorldEJBWS</port-component-name>

                <!--
                    Plain SSL and mutual authentication both require CONFIDENTIAL
                    transport-guarantee to be configured.
                -->
                <transport-guarantee>CONFIDENTIAL</transport-guarantee>
            </webservice-endpoint>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

- Redeploy the web service to GlassFish.

Modifying the Client

As with the client of the servlet-based web service using SSL this client also needs two modifications to be able to interact with the server; modify the URL of the web service to use HTTPS and add JVM parameters to the launch configuration in order for the client to be able to use the client truststore we created earlier.

- Modify the service URL in the generated class *HelloWorldEJBWSService* to use HTTPS (marked with red).

```
@WebServiceClient(name = "HelloWorldEJBWSService", targetNamespace = "http://ivan.com/",
wsdlLocation = "http://localhost:8080/HelloWorldEJBWSService/HelloWorldEJBWS?wsdl")
public class HelloWorldEJBWSService
    extends Service
{

    private final static URL HELLOWORLDEJBWSSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new
URL("https://localhost:8181/HelloWorldEJBWSService/HelloWorldEJBWS?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDEJBWSSERVICE_WSDL_LOCATION = url;
    }
```

- Add the following VM arguments to the launch configuration of the client:
  -Djavax.net.ssl.trustStore=client_cacerts.jks
  -Djavax.net.ssl.trustStorePassword=changeit
  -Djavax.net.debug=all

With these changes in place and after having deployed the modified web service, the client will produce a result similar to what we already have seen. As with the servlet-based web service using SSL, there will be logging output showing the SSL handshaking between the client and server.

Mutual Authentication

When applying the modifications to add mutual authentication to the web service, please start from the web service and client that does not have any security added, as described above.

Configuring the Server

To configure the EJB-based web service to use mutual authentication, perform the following modifications:

- Modify the sun-ejb.xml deployment descriptor to look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB
3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
    <enterprise-beans>
        <ejb>
            <ejb-name>HelloWorldEJBWS</ejb-name>
            <webservice-endpoint>
                <port-component-name>HelloWorldEJBWS</port-component-name>

                <!--
                    For mutual authentication, this element must be
                    configured to use the CLIENT-CERT authentication
                    method. The realm must be certificate, at
                    least in this basic example on GlassFish.
                -->
                <login-config>
                    <auth-method>CLIENT-CERT</auth-method>
                    <realm>certificate</realm>
                </login-config>

                <!--
                    Plain SSL and mutual authentication both require CONFIDENTIAL
                    transport-guarantee to be configured.
                -->
                <transport-guarantee>CONFIDENTIAL</transport-guarantee>
            </webservice-endpoint>
        </ejb>
    </enterprise-beans>
</sun-ejb-jar>
```

- Redeploy the web service to GlassFish.

Modifying the Client

As with the client using SSL, this client also needs two modifications to be able to interact with the server; modify the URL of the web service to use HTTPS and add JVM parameters to the launch configuration in order for the client to be able to use the client truststore we created earlier.

- Modify the service URL in the generated class *HelloWorldEJBWSService* to use HTTPS (marked with red).

```
@WebServiceClient(name = "HelloWorldEJBWSService", targetNamespace = "http://ivan.com/",
wsdlLocation = "http://localhost:8080/HelloWorldEJBWSService/HelloWorldEJBWS?wsdl")
public class HelloWorldEJBWSService
    extends Service
{

    private final static URL HELLOWORLDEJBWSSERVICE_WSDL_LOCATION;

    static {
        URL url = null;
        try {
            url = new
URL("https://localhost:8181/HelloWorldEJBWSService/HelloWorldEJBWS?wsdl");
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
        HELLOWORLDEJBWSSERVICE_WSDL_LOCATION = url;
    }
```

- Add the following VM arguments to the launch configuration of the client:
  -Djavax.net.ssl.trustStore=client_cacerts.jks
  -Djavax.net.ssl.trustStorePassword=changeit
  -Djavax.net.ssl.keyStore=client_keystore.jks
  -Djavax.net.ssl.keyStorePassword=changeit
  -Djavax.net.debug=all

With these changes in place and after having deployed the modified web service, the client will produce a result similar to what we already have seen. Additionally there will be quite some logging output showing the SSL handshaking between the server and client.
To verify that mutual authentication really has taken place, search for the string “CertificateRequest” in the SSH log.

8.4 Web Service Security Factors

Describe factors that impact the security requirements of a Web service, such as the relationship between the client and service provider, the type of data being exchanged, the message format, and the transport mechanism.

In this section, some examples of factors that have impact on the security requirements of web serves are discussed.

Relationship Between Client and Service Provider

The following are examples of relationships between a service provider and clients that will have an impact on security requirements of web services:

- Geographical distribution of the service provider and its clients.
- Whether service provider and clients are belonging to the same organization or not.
- Whether the service provider will bill clients for the use of the service or not.
- Whether the clients connect directly to the service provider or not.
  Alternatively, data may be transmitted through a number of intermediaries.

Type of Data Exchanged

The following are examples of data exchanged that will have an impact on security requirements of web services:

- Publicly available data, such as results of searches in a public library.
- Data related to financial transactions, such as credit card numbers etc.
- Messages with attachment data, perhaps where the attachment is of significant size.

Message Formats

The following are examples of message formats that will have an impact on security requirements of web services:

- SOAP
- Raw XML
- JSON
- If the web service must support multiple message formats.

Transport Mechanisms

The following are examples of transport mechanisms that will have an impact on security requirements of web services:

- JMS (Java Messaging Service)
- HTTP
- SMTP (mail)

8.5 WS-Policy

Describe WS-Policy that defines a base set of constructs that can be used and extended by other Web specifications to describe a broad range of service requirements and capabilities.

References:
http://www.w3.org/TR/ws-policy/
http://www.w3.org/TR/ws-policy-primer/

What is WS-Policy?

From the WS-Policy 1.5 specification document:
“Web Services Policy 1.5 - Framework defines a framework and a model for expressing policies that refer to domain-specific capabilities, requirements, and general characteristics of entities in a Web services-based system.”
Web Services Policy is a language used to express capabilities and requirements of a web service. Several different kinds of policy assertions exists, some examples are:

- Web Services Security Policy.
- Web Services Reliable Messaging Policy.
- Web Services Atomic Transaction Policy.
- Web Services Business Activity Framework.
- Devices Profile for Web Services.

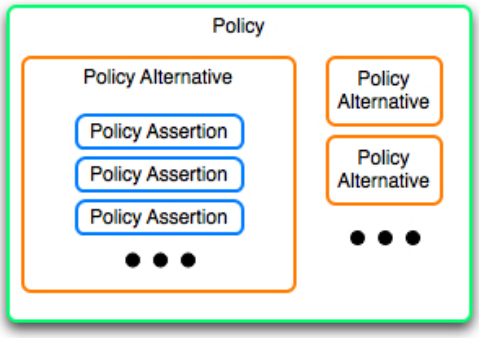
WS-Policy is also extensible by design, allowing it to be extended with new or modified constructs.

Basic Constructs

The basic constructs in the WS-Policy language are:

- Policy
  A collection of policy alternatives.
- Policy Alternative
  A collection of policy assertions.
- Policy Assertion
- Policy Expression
  The XML infoset representation of a policy.

The arrangements of the above constructs can be seen in the following figure:



Note! The above figure does not correspond to the policy expression below, it just shows the containing hierarchy and colour coding!

```
<wsp:Policy
      xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
      xmlns:wsp="http://www.w3.org/ns/ws-policy" >
   <wsp:ExactlyOne>

      <wsp:All>
         <sp:SignedParts>
            <sp:Body/>
         </sp:SignedParts>
      </wsp:All>

      <wsp:All>
         <sp:EncryptedParts>
            <sp:Body/>
         </sp:EncryptedParts>
      </wsp:All>

   </wsp:ExactlyOne>
</wsp:Policy>
```

The policy declared in the above figure can be interpreted to say that an invocation of a web service must either sign or encrypt the body of the message.

Policy Assertions

Policy assertions describe either required behaviour or capabilities of entities such as web service endpoints, web service messages, resources, operations etc.
Behaviour or capabilities, for instance related to security of a web service, indicated by a policy assertion are to be defined in separate specifications.

Policy Alternatives

A policy alternative is a collection of zero or more policy assertions. The collection is unordered and may contain duplicates.

Policy

A policy is a collection of zero or more policy alternatives. The collection is unordered.

Policy Expression

A policy expression is an XML representation of a policy. A policy expression may be in either normal or compact form. The WS-Policy language is very simple, it contains the following four elements and two attributes:

Element/Attribute	Description
<wsp:Policy>	Root element of a policy specification. Can also be used as a policy assertion combining operation, in which case it is equivalent to the <wsp:All> element.
<wsp:All>	Combining multiple policy assertions so that all assertions are required.
<wsp:ExactlyOne>	Combining multiple policy assertions so that exactly one is required.
<wsp:PolicyReference>	Enables re-use of policy declarations by including them in new policy declarations.
wsp:Optional	Specifies whether the policy assertion is optional or not. Default is false.
wsp:Ignorable	Specifies whether the policy assertion can be ignored or not. Default is false.

Normal Form vs Compact Form

The schema outline of the normal form of a policy expression is:

```
<wsp:Policy … >
    <wsp:ExactlyOne>
        ( <wsp:All> ( <Assertion …> … </Assertion> )* </wsp:All> )*
    </wsp:ExactlyOne>
</wsp:Policy>
```

The schema outline of the compact form of a policy expression is:

```
<Assertion ( wsp:Optional="xs:boolean" )? …> … </Assertion>
```

If the value of the *wsp:Optional* attribute is true, then the above is equal to the following in normal form:

```
<wsp:ExactlyOne>
    <wsp:All> <Assertion …> … </Assertion> </wsp:All>
    <wsp:All />
</wsp:ExactlyOne>
```

If the value of the *wsp:Optional* attribute is false, then the above is equal to the following in normal form:

```
<wsp:ExactlyOne>
    <wsp:All> <Assertion …> … </Assertion> </wsp:All>
</wsp:ExactlyOne>
```

Attaching Policies to WSDL Documents

Policy expressions can be attached to <binding> elements in WSDL documents, either by inserting the policy declaration or by referencing it.

Example:

The first example shows a policy expression having been attached to a <binding> element in a WSDL document by inserting the entire policy expression in the <binding> element.

```
…
<wsdl:binding name="MyBinding" type="tns:MyInterface" >
    <wsp:Policy>
        <wsam:Addressing>…</wsam:Addressing>
    </wsp:Policy>
    …
</wsdl:binding>
…
```

The second example shows a policy expression having been attached to a <binding> element in a WSDL document by using the <PolicyReference> element. The policy is declared elsewhere.

```
…
<wsdl:binding name="InquiryBinding" type="tns:InquiryInterface" >
    <wsp:PolicyReference URI="#common" />
    <wsdl:operation name="MakeInquiry">…</wsdl:operation>
    …
</wsdl:binding>
…
```

Additional Examples

Example:

The following compact form policy expression:

```
<wsp:Policy
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:wsp="http://www.w3.org/ns/ws-policy" >
    <sp:IncludeTimestamp wsp:Optional="true" />
</wsp:Policy>
```

Is equivalent to the following normal form policy expression:

```
<wsp:Policy
    xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
    xmlns:wsp="http://www.w3.org/ns/ws-policy" >
    <wsp:ExactlyOne>
        <wsp:All>
            <sp:IncludeTimestamp />
        </wsp:All>
        <wsp:All />
    </wsp:ExactlyOne>
</wsp:Policy>
```

Example:
The following policy expression requires:

- Addressing.
- Optionally MIME-serialization.
- Transport or message-level security.

Additionally there is a policy assertion informing clients that the service performs logging of some kind. This policy assertion may be completely ignored, as specified by the *wsp:Ignorable* attribute.

```
<All>
    <mtom:OptimizedMimeSerialization wsp:Optional="true"/>
    <wsam:Addressing>…</wsam:Addressing>
    <log:Logging wsp:Ignorable="true" />
    <ExactlyOne>
        <sp:TransportBinding>…</sp:TransportBinding>
        <sp:AsymmetricBinding>…</sp:AsymmetricBinding>
    </ExactlyOne>
</All>
```

Example:
The second policy re-uses the first policy declaration.

```
<Policy xml:id=”common”>
    <mtom:OptimizedMimeSerialization wsp:Optional="true"/>
    <wsam:Addressing>…</wsam:Addressing>
</Policy>
...
<Policy wsu:Id=”secure”>
    <All>
        <PolicyReference URI="#common"/>
        <ExactlyOne>
            <sp:TransportBinding>…</sp:TransportBinding>
            <sp:AsymmetricBinding>…</sp:AsymmetricBinding >
        </ExactlyOne>
    </All>
</Policy>
```

9. Developing Web Services

References:
http://www.developer.com/java/ent/article.php/3730756
http://jcp.org/en/jsr/detail?id=181
http://jcp.org/en/jsr/detail?id=109

9.1 Configuration, Packaging and Deployment

Describe the steps required to configure, package, and deploy Java EE Web services and service clients, including a description of the packaging formats, such as .ear, .war, .jar, annotations and deployment descriptor settings.

Note!
Only configuration, packaging and deployment of JAX-WS web services has been considered.

First a word on terminology: JSR-109 uses the name *port component* as the name of a component that is packed and deployed to a container in order to implement a web service.

Configuration of Web Services

References:
Web Services for JavaEE v1.2 (JSR-109), section 5.3, chapter 7.

Web services can be configured in the following different ways:

- Annotations.
  Common for both servlet and EJB based web services. Please refer to the section on web service annotations above for a detailed descriptions of the different annotations.
  Configuration of data binding, handled by JAXB, is also done using annotations.
- The webservices.xml deployment descriptor.
  Also common for both servlet and EJB based web services. Overlaps configuration with annotations and can be used to override or augment the annotation configurations.
  Not required.
- The web.xml deployment descriptor.
  Configuration specific to servlet based web services. Not required.
- The ejb-jar.xml deployment descriptor.
  Configuration specific to EJB based web services. Not required.

Annotations and the webservices.xml Deployment Descriptor

If using the webservices.xml, the developer of a web service is responsible for specifying the following things:

1. Logical name of the port component.
   Not related to WSDL port names.
   Name must be unique within all port component names in a module.
2. Port component's bean implementation class.
3. Port component's service endpoint interface.
4. Location of the port component's WSDL file.
   Either a file location relative to the module root or an URL. Optional.
5. WSDL Service QName.
   Required if the bean implementation class implements the JAX-WS *Provider<T>* interface, optional otherwise.
6. WSDL Port QName.
   QName for each port defined in the port component's WSDL file.
7. MTOM/XOP Support.
   Enable or disable MTOM/XOP support for the port component. Optional.
8. Protocol binding.
   Overrides protocol binding specified by the @BindingType annotation. Default is SOAP 1.1/HTTP. Optional.
9. Handlers.
   Specify one or more handlers associated with the port component. Optional.

The following is an example of a webservices.xml deployment descriptor in which the above items have been enumerated.

```
<webservices xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.2"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/javaee_web_services_1_2.xsd">

    <!--
         This element defines a WSDL document file and a set of port
         components that are associated to the ports defined in the
         WSDL document.
         A module (JAR/WAR-file) may contain multiple webservice
         descriptions.
    -->
    <webservice-description>
        <webservice-description-name>HelloService</webservice-description-name>

        <!--
            Item 4.
            Specifies the location of the WSDL file relative to the root of
            the module (JAR/WAR-file).
            Corresponding annotations:
            @WebService.wsdlLocation, @WebServiceProvider.wsdlLocation
        -->
        <wsdl-file>WEB-INF/wsdl/hello-service.wsdl</wsdl-file>

        <!--
            Each web service class, either a plain web service implementation class
            or a web service provider class, is a port component.
            Corresponding annotations: @WebService, @WebServiceProvider.
        -->
        <port-component>
            <!--
                Item 1.
                Port component name corresponds to either:
                - @WebService.name
                - Fully qualified name of endpoint implementation class.
```

```
        Must be unique within the module. If @WebService.name is not unique,
        then the fully qualified class name will be used, which also is
        the default value.
        Note! Not related to WSDL port name.
        Corresponding annotations: @WebService.name
    -->
    <port-component-name>Hello</port-component-name>

    <!--
        Item 5.
        Name of the <wsdl:service> element of this port component.
        Required if the port component is a web service provider.
        Corresponding annotations:
        @WebService.serviceName or @WebServiceProvider.serviceName
    -->
    <wsdl-service xmlns:ns1="http://www.ivan.com/">
        ns1:HelloService
    </wsdl-service>

    <!--
        Item 6.
        Name of the <wsdl:port> element of this port component.
        Corresponding annotations:
        @WebService.portName or @WebServiceProvider.portName
    -->
    <wsdl-port xmlns:ns1="http://www.ivan.com/">ns1:HelloPort</wsdl-port>

    <!--
        Item 7.
        Specify whether MTOM/XOP support is enabled or not.
    -->
    <enable-mtom>false</enable-mtom>

    <!--
        Item 8.
        Specifies the protocol binding for the port component.
        Default is SOAP 1.1 over HTTP.
        The following predefined tokens are available:
        ##SOAP11_HTTP, ##SOAP12_HTTP, ##SOAP11_HTTP_MTOM,
        ##SOAP12_HTTP_MTOM, ##XML_HTTP.
    -->
    <protocol-binding>##SOAP11_HTTP</protocol-binding>

    <!--
        Item 3.
        Specifies the service endpoint interface, if one is used.
        Corresponds to @WebService.endpointInterface.
        May not be used when there is no service endpoint interface.
        If the service implementation is a stateless EJB, then the
        service endpoint interface must also be specified in the ejb-jar.xml
        deployment descriptor.
        Corresponding annotations: @WebService.endpointInterface
    -->
    <service-endpoint-interface>
        com.ivan.HelloInterface
    </service-endpoint-interface>

    <!--
        Item 2.
        Defines the web service implementation by specifying either a link
        to a servlet or a link to an EJB.
    -->
    <service-impl-bean>
        <!--
            For a servlet endpoint, the <servlet-link> element is used and
            contains the name of the servlet defined in the web.xml
            deployment descriptor that implements the service endpoint.
            For an EJB endpoint, the <servlet-link> is replaced with an
            <ejb-link>, specifying the name of the session bean implementing
            the service endpoint.
            The EJB must be located in the same JAR/WAR-file as the
            webservices.xml deployment descriptor is located in.

            A servlet and an EJB may only be linked to by one single port
            component.
        -->
        <servlet-link>Hello</servlet-link>
```

```
            </service-impl-bean>

            <!--
                Item 9.
                Handlers.
            -->
            <handler-chains>
                <handler-chain>
                    <handler>
                        <handler-name>someHandler</handler-name>
                        <handler-class>com.ivan.MyHandler</handler-class>
                    </handler>
                </handler-chain>
            </handler-chains>
        </port-component>
    </webservice-description>
</webservices>
```

The following table describe the relationships between elements of the webservices.xml deployment descriptor and the JAX-WS @WebService annotation.

Deployment Descriptor Element	WebService Annotation
<webservice-description>	One per WSDL document.
<port-component>	One per @WebService annotation.
<wsdl-file>	@WebService.wsdlLocation
<port-component-name>	@WebService.name
<wsdl-service>	@WebService.serviceName
<wsdl-port>	@WebService.portName
<service-endpoint-interface>	@WebService.endpointInterface

The following table describes the relationships between elements of the webservices.xml deployment descriptor and the JAX-WS @WebServiceProvider annotation.

Deployment Descriptor Element	WebServiceProvider Annotation
<webservice-description>	One per WSDL document.
<port-component>	One per @WebServiceProvider annotation.
<wsdl-file>	@WebServiceProvider.wsdlLocation
<port-component-name>	Only used to guarantee uniqueness of the port component.
<wsdl-service>	@WebServiceProvider.serviceName
<wsdl-port>	@WebServiceProvider.portName
<service-endpoint-interface>	Not used.

Additionally, the <handler-chains> element in the webservices.xml deployment descriptor corresponds to the @HandlerChain annotation.

Servlet-Based Web Service Configuration Example

The following is an example showing:

- How a web service endpoint interface is annotated.
- How a web service servlet implementation bean is annotated.
- What a webservices.xml deployment descriptor matching the above look like.
- What a web.xml deployment descriptor matching the above look like.
- The WSDL of the web service.

The web service endpoint interface declaration:

```
...
@WebService
public interface HelloWorldWSInterface
{
    ...
```

The web service implementation bean declaration and instance variable declarations:

```
...
@WebService(
    endpointInterface = "com.ivan.HelloWorldWSInterface",
    wsdlLocation = "WEB-INF/wsdl/HelloWorldWSService.wsdl",
    targetNamespace = "http://ivan.com/",
    portName = "HelloWorldWSInterface",
    serviceName = "HelloWorldWSService")
public class HelloWorldWS implements HelloWorldWSInterface
{
    @Resource
    private WebServiceContext mWSContext;
    ...
```

The complete webservices.xml deployment descriptor.
Note that not all declared properties of the port component in this example are required.

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.2"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://www.ibm.com/webservices/xsd/javaee_web_services_1_2.xsd">

    <webservice-description>
        <display-name>HelloWorldWSService</display-name>
        <webservice-description-name>HelloWorldWSService</webservice-description-name>
        <wsdl-file>WEB-INF/wsdl/HelloWorldWSService.wsdl</wsdl-file>

        <port-component>
            <port-component-name>HelloWorldWS</port-component-name>
            <wsdl-service xmlns:ns1="http://ivan.com/">
                ns1:HelloWorldWSService
            </wsdl-service>
            <wsdl-port xmlns:ns1="http://ivan.com/">
                ns1:HelloWorldWSInterface
            </wsdl-port>
            <enable-mtom>false</enable-mtom>
            <protocol-binding>##SOAP11_HTTP</protocol-binding>
            <service-endpoint-interface>
                com.ivan.HelloWorldWSInterface
            </service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>HelloWorldWS</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

The complete web.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    metadata-complete="true"
    version="2.5"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <display-name>JAX-WS_HelloWorldServlet</display-name>


    <!-- Web service servlet endpoint. -->
    <servlet>
        <display-name>HelloWorldWS</display-name>
        <servlet-name>HelloWorldWS</servlet-name>
        <servlet-class>com.ivan.HelloWorldWS</servlet-class>
    </servlet>

    <!-- Mapping for the web service servlet endpoint. -->
    <servlet-mapping>
        <servlet-name>HelloWorldWS</servlet-name>
        <url-pattern>/HelloWorldWSService</url-pattern>
    </servlet-mapping>

    <!--
        If we supply a web.xml deployment descriptor and also want
        the web service context to be injected into the mWSContext
        instance variable in the web service implementation bean,
        then we need, beside the @Resource annotation, the following:
    -->
    <resource-ref>
        <res-ref-name>com.ivan.HelloWorldWS/mWSContext</res-ref-name>
        <res-type>javax.xml.ws.WebServiceContext</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
        <injection-target>
            <injection-target-class>com.ivan.HelloWorldWS</injection-target-class>
            <injection-target-name>mWSContext</injection-target-name>
        </injection-target>
    </resource-ref>
</web-app>
```

Additionally, the WSDL file of the web service looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://ivan.com/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://ivan.com/"
    name="HelloWorldWSService">

    <types>
        <xsd:schema>
            <xsd:import
                namespace="http://ivan.com/"
                schemaLocation="HelloWorldWSTypes.xsd"/>
        </xsd:schema>
    </types>
    <message name="hello">
        <part name="parameters" element="tns:hello"/>
    </message>
    <message name="helloResponse">
        <part name="parameters" element="tns:helloResponse"/>
    </message>

    <portType name="HelloWorldWSInterface">
        <operation name="hello">
            <input message="tns:hello"/>
            <output message="tns:helloResponse"/>
        </operation>
```

```
    </portType>

    <binding name="HelloWorldWSInterfaceBinding" type="tns:HelloWorldWSInterface">
        <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
        <operation name="hello">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="HelloWorldWSService">
        <port name="HelloWorldWSInterface" binding="tns:HelloWorldWSInterfaceBinding">
            <soap:address location="TBA"></soap:address>
        </port>

    </service>
</definitions>
```

As an exercise to show that the webservices.xml deployment descriptor indeed overrides the configuration supplied in the web service implementation bean class, do the following:

- Create another endpoint interface with the exact same contents as the original one, but with another name.
- Modify the contents of the <service-endpoint-interface> tag in the webservices.xml deployment descriptor to use the new endpoint interface.
- Deploy the web service.
- In GlassFish, examine the webservices.xml deployment descriptor of the web service.
- In the same way, the WSDL for a web service can be exchanged for an alternate WSDL by modifying the webservices.xml deployment descriptor.

When developing on GlassFish v2 and supplying a web.xml deployment descriptor but no webservices.xml deployment descriptor, a servlet-based web service will not be recognized as a web service. After having supplied a webservices.xml deployment descriptor, the web service will once again appear in GlassFish.

EJB-Based Web Service Configuration Example

The following is an example showing:

- How a web service endpoint interface is annotated.
- How a web service EJB implementation bean is annotated.
- What a webservices.xml deployment descriptor matching the above look like.
- What a ejb-jar.xml deployment descriptor matching the above look like.
- The WSDL file of the web service.

The web service endpoint interface, annotated in the same way as with a servlet-based web service.

```
...
@WebService
public interface GreetingServiceInterface
{
    String greeting(final String inMessage);
}
```

The web service implementation bean declaration and instance variable declarations:

```
...
@Stateless(name="GreetingServiceEJB")
@WebService(
    endpointInterface="com.ivan.GreetingServiceInterface",
    wsdlLocation="META-INF/wsdl/GreetingService.wsdl",
    targetNamespace="http://ivan.com/",
    portName="GreetingServiceEJBPort",
    serviceName="GreetingServiceEJBService")
public class GreetingServiceEJB implements GreetingServiceInterface
{
    @Resource
    private WebServiceContext mWSContext;
    ...
```

The complete webservices.xml deployment descriptor.
Note that not all declared properties of the port component in this example are required.

```
<?xml version="1.0" encoding="UTF-8"?>
<webservices
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="1.2"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://www.ibm.com/webservices/xsd/javaee_web_services_1_2.xsd">

    <webservice-description>
        <display-name>GreetingServiceEJBService</display-name>
        <webservice-description-name>
            GreetingServiceEJBService
        </webservice-description-name>
        <wsdl-file>META-INF/wsdl/GreetingService.wsdl</wsdl-file>

        <port-component>
            <port-component-name>GreetingServiceEJB</port-component-name>
            <wsdl-service xmlns:ns1="http://ivan.com/">
                ns1:GreetingServiceEJBService
            </wsdl-service>
            <wsdl-port xmlns:ns1="http://ivan.com/">
                ns1:GreetingServiceEJBPort
            </wsdl-port>

            <!--
                If we want to specify a service endpoint interface here,
                then we also need to define it as a business interface of
                the EJB in the ejb-jar.xml deployment descriptor.
            -->
            <service-endpoint-interface>
                com.ivan.GreetingServiceInterface
            </service-endpoint-interface>
```

```
            <service-impl-bean>
                <ejb-link>GreetingServiceEJB</ejb-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

The complete ejb-jar.xml deployment descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    metadata-complete="true" version="3.0"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">

    <enterprise-beans>
        <!-- The stateless session bean implementing the web service. -->
        <session>
            <display-name>GreetingServiceEJB</display-name>
            <ejb-name>GreetingServiceEJB</ejb-name>

            <!--
                If we want to use a service endpoint interface for
                the web service implemented by the EJB, then we need to
                define the interface here too.
            -->
            <business-local>com.ivan.GreetingServiceInterface</business-local>

            <!--
                If we have a service endpoint interface, then is fully
                qualified name must be used in the <service-endpoint>
                element below.
            -->
            <service-endpoint>com.ivan.GreetingServiceInterface</service-endpoint>
            <ejb-class>com.ivan.GreetingServiceEJB</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>

            <!--
                If we supply a ejb-jar.xml deployment descriptor and also
                want the web service context to be injected into the
                mWSContext instance variable in the web service
                implementation bean, then we need, beside the @Resource
                annotation, the following:
            -->
            <resource-ref>
                <res-ref-name>com.ivan.GreetingServiceEJB/mWSContext</res-ref-name>
                <res-type>javax.xml.ws.WebServiceContext</res-type>
                <res-auth>Container</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
                <injection-target>
                    <injection-target-class>
                        com.ivan.GreetingServiceEJB
                    </injection-target-class>
                    <injection-target-name>mWSContext</injection-target-name>
                </injection-target>
            </resource-ref>

            <!-- Security identity configuration for the EJB. -->
            <security-identity>
                <use-caller-identity />
            </security-identity>
        </session>
    </enterprise-beans>
</ejb-jar>
```

Finally, the WSDL file of the web service:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
    xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://ivan.com/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://ivan.com/"
    name="GreetingServiceEJBService">

    <types>
        <xsd:schema>
            <xsd:import
                namespace="http://ivan.com/"
                schemaLocation="GreetingServiceTypes.xsd"/>
        </xsd:schema>
    </types>

    <message name="greeting">
        <part name="parameters" element="tns:greeting"></part>
    </message>
    <message name="greetingResponse">
        <part name="parameters" element="tns:greetingResponse"></part>
    </message>

    <portType name="GreetingServiceInterface">
        <operation name="greeting">
            <input message="tns:greeting"></input>
            <output message="tns:greetingResponse"></output>
        </operation>
    </portType>

    <binding name="GreetingServiceEJBPortBinding" type="tns:GreetingServiceInterface">
        <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
        <operation name="greeting">
            <soap:operation soapAction=""/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="GreetingServiceEJBService">
        <port name="GreetingServiceEJBPort" binding="tns:GreetingServiceEJBPortBinding">
            <soap:address location="TBA"/>
        </port>
    </service>
</definitions>
```

When developing on GlassFish v2 and supplying a ejb-jar.xml deployment descriptor but no webservices.xml deployment descriptor, an EJB-based web service will not be recognized as a web service. After having supplied a webservices.xml deployment descriptor, the web service will once again appear in GlassFish.

Configuration of Web Service Clients

References:
Java Servlet Specification v2.4, chapter 13.
EJB 3.0 Core Specification, section 16.6.
JAX-WS 2.1 Specification
Web Services for JavaEE v1.2 (JSR-109)

The developer of a web services client is responsible for defining a reference for each web service the client wants to use. A web service reference includes the following information:

- Service reference name.
  The JNDI name the web service reference should be mapped to. It is recommended that it starts with “service/”.
- Service type.
  The fully qualified name of the service interface or class returned by looking up the service reference JNDI name.
  This is usually the service object from which a port is then to be retrieved.
- Service reference type.
  The fully qualified name of the type that is injected or returned by the JNDI lookup.
  This must be either a fully qualified name of the javax.xml.ws.Service class or the fully qualified name of a service endpoint interface.
  This is usually the type of the port retrieved from a service object.
- Ports.
  Requirements for container managed port resolution.
- MTOM/XOP Support.
  Enable or disable MTOM/XOP support for the referenced web service.
- WSDL file location.
- Service.
  If the WSDL file has more than one <wsdl:service> element, then the qualified name of the service element to use must be specified.
- Handlers.
  Optionally, handlers associated with the referenced web service can be specified.

Web service clients can be configured in the following different ways:

- Annotations.
  Usable by all kinds of client.
- In the web.xml deployment descriptor.
  For web service clients that are servlets.
- In the ejb-jar.xml deployment descriptor.
  For web service clients that are EJBs.
- In the application-client.xml deployment descriptor.
  For standalone applications.
- Programmatically.
  Please refer to the section on Dynamic Clients above.

Using Annotations

Annotation-based configuration can be used with both JavaEE and standalone web service clients. However, if being used with a standalone web service client, then the client must be run in some kind of container or managed environment that interprets annotations and manages dependency injection. Running standalone clients in a container is considered to be outside the scope of this document. For GlassFish, please refer to the *appclient* script included with the server.

The javax.xml.ws.WebServiceClient Annotation

The @WebServiceClient annotation is used to annotate a class, typically generated, extending the *javax.xml.ws.Service* class. The information specified in the annotation binds the service class to a particular <wsdl:service> element in a specified WSDL file. The annotation contains the following elements:

Element Name	Description
name	Local name of the <wsdl:service> element to which the service class is to be bound.
targetNamespace	The target namespace of the WSDL document in which the <wsdl:service> element is located.
wsdlLocation	URL specifying the location of the WSDL document in which the <wsdl:service> element is located.

There is seldom need to manually add this annotation, since stub classes generated by the *wsimport* tool are automatically annotated as needed.
Example use:

```
@WebServiceClient(
    name = "CalculatorService",
    targetNamespace = "http://www.ivan.com/calculator",
    wsdlLocation = "http://localhost:8080/JAX-WS_Server_wsgen/CalculatorService?wsdl")
public class CalculatorService extends Service
{
```

The javax.xml.ws.WebServiceRef Annotation

The @WebServiceRef annotation is used to declare a reference to a web service. Classes, methods or fields can be annotated. The JAX-WS specification only require this annotation to be honored when running on the JavaEE 5 platform.
The annotation contains the following elements:

Element Name	Description
mappedName	A name this resource should be mapped to. This name is often a global JNDI name. Containers are not required to support any particular form or type of mapped name. Default: “”
name	JNDI name this resource should be mapped to. Default: “”
type	The fully qualified name of the service interface or class returned by looking up the service reference JNDI name. Default: Object.class
value	Java type of the service. Default: Object.class
wsdlLocation	URL at which WSDL document of the web service can be found.

Example use:

```
@WebServiceRef(
    name = "myService",
    value = MyEndpointService.class,
    type = MyEndpoint.class
    wsdlLocation = "META-INF/wsdl/MyEndpoint.wsdl")
public class MyClient
{
...
```

Also see the example in the next section on the @WebServiceRefs annotation.

The javax.xml.ws.WebServiceRefs Annotation

The @WebServiceRefs annotation is used to declare multiple references to web services. Only classes can be annotated and, since the name and type cannot be inferred, each @WebServiceRef annotation contained in the @WebServiceRefs annotation must specify both name and type.
The annotation contains the following element:

Element Name	Description
values	Array of web service reference declarations.

Example use:

```
@WebServiceRefs({
    @WebServiceRef(
        name="service/MyHelloService",
        type=servlet_endpoint.MyHelloService.class,
        wsdlLocation="http://localhost:8080/wsrefs/ws/MyHelloService?WSDL"),
    @WebServiceRef(
        name="service/MyEjbService",
        type=ejb_endpoint.HelloEJBService.class,
        wsdlLocation="http://localhost:8080/HelloEJBService/Hello?WSDL") })
public class Client
{
...
    public void useWebService()
    {
        try
        {
            InitialContext ic = new javax.naming.InitialContext();
            MyHelloService service =
                (MyHelloService)ic.lookup("java:comp/env/service/MyHelloService");
                MyHello port = service.getMyHelloPort();
            ...
```

The javax.jws.HandlerChain Annotation

The @HandlerChain annotation can not only be used on web service servers, but also on web service clients. The following locations in a web service client can be annotated:

- On the service class.
  All proxies created by the service will have their handler chain configured according to the configuration file specified in the annotation.
- On web service references.
  The injected service will have a handler chain configured according to the configuration file specified in the annotation.

Example of annotating a service class:

```
@WebServiceClient(
    name="HelloService",
    targetNamespace="http://example.com/handlers",
    wsdlLocation="http://localhost:8080/service/HelloService?wsdl")
@HandlerChain(file="HelloService_handler.xml")
public class HelloService extends Service
{
   @WebEndpoint
   ...
   ...
}
```

Example of annotating a web service reference:

```
public class WebClient extends HttpServlet
{
    @HandlerChain(file="myhandler.xml")
    @WebServiceRef HelloService service;

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws javax.servlet.ServletException
    {
       ...
    }
}
```

Using Deployment Descriptors

All three deployment descriptors, web.xml, ejb-jar.xml and application-client.xml, declares references to web services in <service-ref> elements. Configuration in a deployment descriptor overrides any configuration in annotations.

> **Note!** The following is a fragment of a deployment descriptor showing how to declare a reference to a service. It is not an actual deployment descriptor, but is intended to list all the different configuration elements in the <service-ref> element.

```
<!-- Declare one reference to a web service. -->
<service-ref>
    <!--
        Define the JNDI name by which the reference to the service
        may be looked up.
    -->
    <service-ref-name>services/GreetingService</service-ref-name>

    <!--
        The fully qualified name of the service interface or class
        returned by looking up the service reference JNDI name.
        This is not a Java interface, but either the javax.xml.ws.Service
        class or a subclass of this class.
    -->
    <service-interface>com.ivan.service.GreetingService</service-interface>

    <!--
        The fully qualified name of the type that is injected or
        returned by the JNDI lookup.
        This must be either a fully qualified name of the
        javax.xml.ws.Service class or the fully qualified name of
        a service endpoint interface.
        Default is the type specified by the <service-interface>.
    -->
    <service-ref-type>com.ivan.service.Greeting</service-ref-type>

    <!--
        URI location of the WSDL file of the web service.
        If WSDL contained in module, then relative to the root of
        the module. Ex: WEB-INF/wsdl/GreetingService.wsdl
        WSDL may also be external to the module, in which case
        the URL of the WSDL document is to be supplied.
    -->
    <wsdl-file>
        http://localhost:8080/JAX-WS_GreetingService/GreetingService?wsdl
    </wsdl-file>

    <!--
        Qualified name of the <wsdl:service> element to which the
        service reference is to be bound.
    -->
    <service-qname xmlns:ik="http://ivan.com/greeting">
        ik:GreetingService
    </service-qname>

    <!--
        Associates a port component with a Service Endpoint Interface.
    -->
    <port-component-ref>
        <!--
            Fully qualified name of the Service Endpoint Interface class
            of the port component.
        -->
        <service-endpoint-interface>
            com.ivan.service.GreetingService
        </service-endpoint-interface>

        <!-- Enable or disable MTOM for the port component. -->
        <enable-mtom>false</enable-mtom>

        <!--
```

```
            Name of the port component, as declared either in the
            <port-component-name> element in a webservices.xml deployment
            descriptor or in a @WebService annotation's name element.
        -->
        <port-component-link>GreetingServicePort</port-component-link>
    </port-component-ref>

    <!-- Handler chain declarations. -->
    <handler-chain id="hcs_123">
        <handler-chain id="hc_1">
            <handler id="h_1">
                ...
            </handler>
            <handler id="h_2">
                ...
            </handler>
        </handler-chain>
        <handler-chain id="hc_2">
            ...
        </handler-chain>
    </handler-chains>
</service-ref>
```

The web service which reference is declared in the above deployment descriptor can then be used by the client in the following fashion:

```
...
Context context= new InitialContext();
Object obj = context.lookup("java:comp/env/services/GreetingService");
com.ivan.service.Greeting service= (com.ivan.service.Greeting)obj;
com.ivan.service.GreetingService port = service.getConverterPort();
...
```

Packaging of Web Services

References:
Web Services for JavaEE v1.2 (JSR-109), sections 5.3-5.4, 7.1.3

A port component, that is a component implementing a web service, can contain the following artifacts:

Artifact	Required	Comments
WSDL Document	No	Describes the web service.
Service Endpoint Interface (SEI)	No	Defines the methods exposed by the web service.
Service Implementation Bean and dependent classes.	Yes	Implements the business logic of the web service. Defines the contract which allows business logic to interact with container services. Implements the same methods with the same signatures as found in the SEI, but is not required to implement the SEI itself.
Deployment Descriptor, containing, among others, Security Role References	No	Security Role References are logical security role names used in the port component.
JAX-WS generated portable artifacts.	No	Zero or more Java bean classes and any service-specific exceptions.
OASIS XML Catalog 1.1	No	Used when resolving WSDL and XML schema documents.

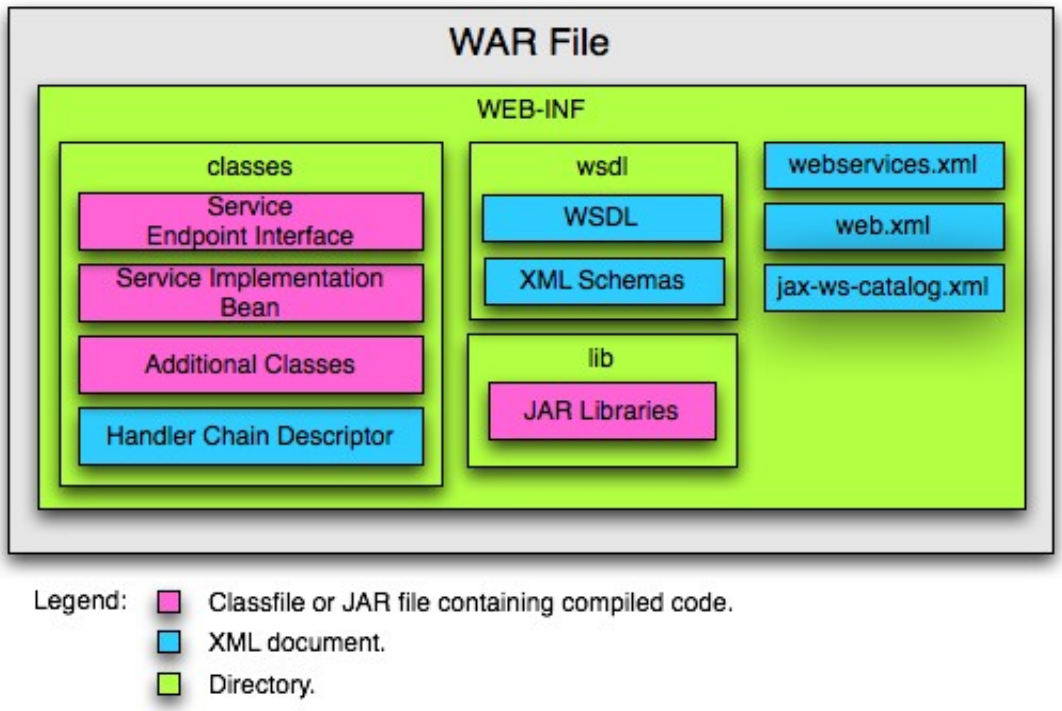
Port components can be packaged in two different ways, depending on what kind of service implementation bean is used:

- In a WAR file.
  Service implementation bean must be a JAX-WS Service Endpoint.
- In a EJB-JAR file.
  Service implementation bean must be a Stateless Session EJB.

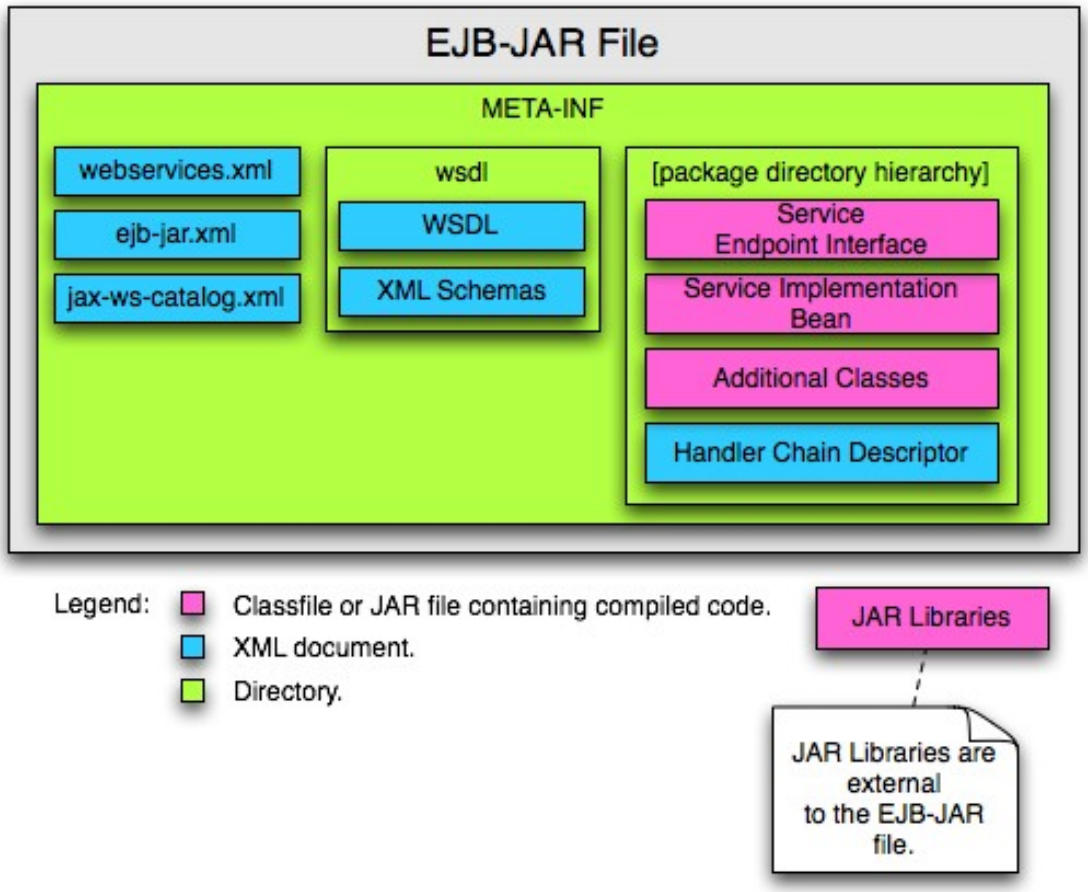
The different artifacts are packed according to the following table:

Artifact	Packaging
WSDL Document	Included in the package or referenced, if included: In WAR: In WEB-INF/wsdl directory. In EJB-JAR: In META-INF/wsdl directory.
Service Endpoint Interface (SEI)	Included in the package.
Service Implementation Bean and dependent classes.	Included in the package.
Deployment Descriptor, containing, among others, Security Role References	In WAR: In WEB-INF/webservices.xml. In EJB-JAR: In META-INF/webservices.xml.
JAX-WS generated portable artifacts.	Included in the package.
OASIS XML Catalog 1.1	In WAR: In WEB-INF/jax-ws-catalog.xml. In EJB-JAR: In META-INF/jax-ws-catalog.xml.

Packaging is illustrated by the following figures showing packaging in WAR, JAR and EAR files.

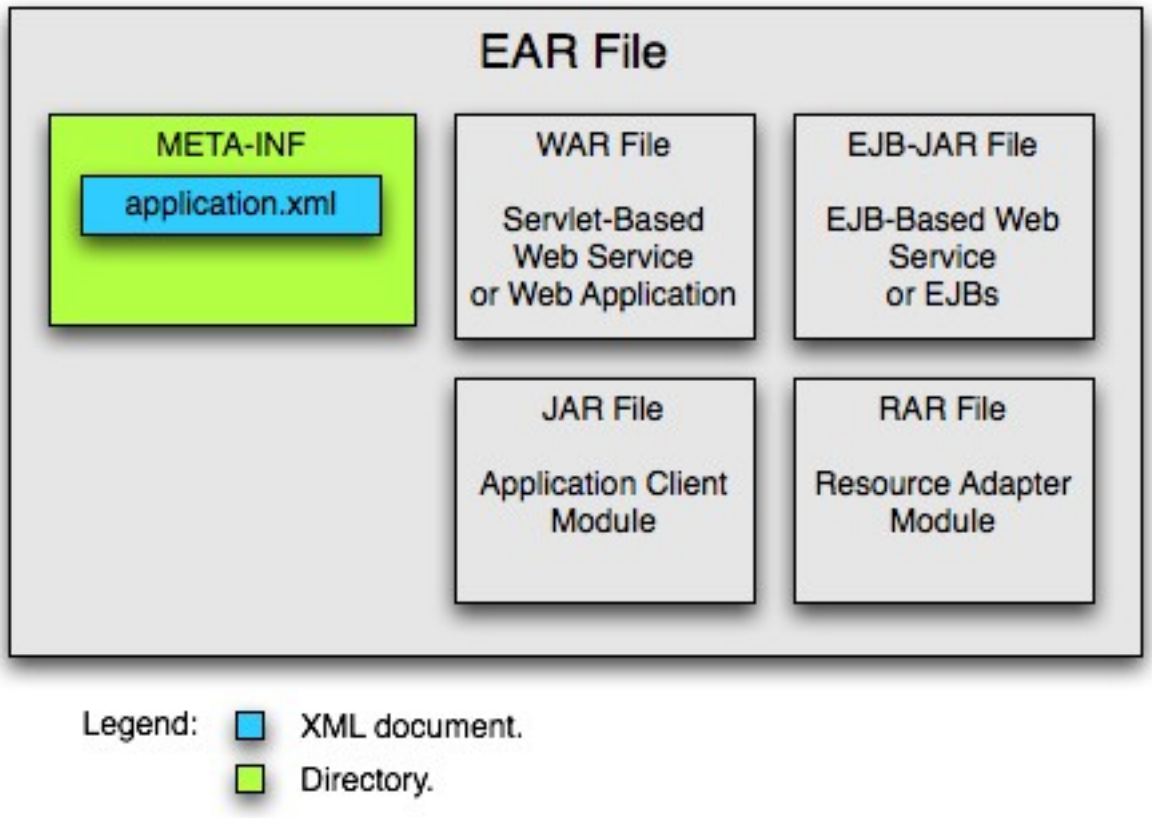


Port component packaging in a WAR file.



Port component packaging in an EJB-JAR file.

WAR files and EJB-JAR files may in turn be assembled into an EAR file:



The assembler of a web service may modify the following things specified by a developer in the webservices.xml deployment descriptor:

- Change or add <description> elements.
- Change values of handler <param-value> elements.
- Add new handler <init-param> elements.
- Change or add handler <soap-role> elements.
- Add new <handler-chain> or <handler> elements.

With JAX-WS, a handler implementation must contain all SOAP header information needed by the handler.
The assembler's basic responsibilities are to:

- Create a deployable artifact by composing multiple modules.
- Resolve cross-module dependencies.
- Providing any configuration that overrides annotation-based configuration.
- Produce an EAR file.

Packaging of Web Service Clients

References:
Web Services for JavaEE v1.2 (JSR-109), sections 4.2.12 and 7.2.3

There are two kind of Java web service client:

- Standalone clients.
  Any kind of client that does not execute in a container is considered to be a standalone client. JavaME web service clients are included in this category.
- JavaEE clients.
  Clients running in a container, such as a web container or an EJB container, are considered to be JavaEE clients.

A web service client includes the following artifacts:

Artifact	Required	Description
Service Endpoint Interface class(es).	Yes	Either included in the client or referenced by being included on the client's classpath.
Generated service interface class.	No	Either included in the client or referenced by being included on the client's classpath.
Additional client class(es).	Yes	Either included in the client or referenced by being included on the client's classpath.
Web service client deployment descriptor.	No, not if configuration done with annotations.	Either in WEB-INF/ for web applications or META-INF/ for EJB modules and standalone clients.
WSDL file(s).	Yes	Either included in the client or referenced by an URL. Location is usually in a wsdl directory with the deployment descriptor.
OASIS XML Catalogs file jax-ws-catalog.xml	No	Located with the module deployment descriptor.

JavaEE web service clients are packaged in the same way as web services; please refer to the previous section for detailed descriptions!
Standalone web service clients are packaged as standard Java applications in a JAR file that contains a META-INF directory in the root of the JAR.

An assembler of a web service client has the following responsibilities, related to <service-ref> elements in the client deployment descriptor:

- May link a web service reference to a component within the JavaEE application unit using the <port-component-link> element.
- Ensure that there are no differences between the service endpoint interface and target bindings that may cause problems generating stubs or at runtime.

Additionally the assembler:

- May replace a WSDL file that resolves missing <service> elements, <port> elements or *address* attributes of <port> elements.

The assembler of a web service client may modify the following things specified by a developer in a module's <service-ref> element:

- Change or add <description> elements.
- Change values of handler <param-value> elements.
- Add new handler <init-param> elements.
- Change or add handler <soap-role> elements.
- Add new <handler-chain> or <handler> elements.

With JAX-WS, a handler implementation must contain all SOAP header information needed by the handler.

Deploying Web Services

References:
Web Services for JavaEE v1.2 (JSR-109), sections 7.1.4, 8.3

A web service can be deployed in three different ways:

- As a servlet endpoint.
  Deployed as a regular web-application in a WAR file.
- As a stateless EJB endpoint.
  Deployed as EJB modules.
- In an enterprise application.
  A web service module in an enterprise application is contained in either a web application module (WAR file) or an EJB module.

Deploying web-applications, EJB modules and enterprise applications are specific to the container used and will not be described in this document.
The deployer of a web service is responsible for specifying deployment time binding information.

- Providing bindings for port addresses of port components in the web service in the WSDL file of the service.
- Provide bindings for container managed port access to Service Endpoint Interfaces.

Deploying Web Service Clients

References:
Web Services for JavaEE v1.2 (JSR-109), sections 7.2.4

JavaEE web service clients are either JavaEE clients or standalone applications. JavaEE web service clients. In addition to the normal responsibilities of a JavaEE deployer, the deployer must also consider the following when deploying web service clients:

- Provide binding information to ensure that each <service-ref> specified by the client can be resolved.
- If a partial WSDL document was specified and <service> and <port> elements are needed to resolve the binding, they may be generated.
- Provide binding information to ensure that <port-component-ref> elements in <service-ref> elements can be resolved.

9.2 XML File Processing

Given a set of requirements, develop code to process XML files using the SAX, StAX, DOM, XSLT, and JAXB APIs.

For motivations when to use respective API, please refer to section 5.5 about the JAXP APIs and section 5.6 about JAXB.

Prerequisites

We will use the following XML schema and an XML document based on that schema in the example code in this section. Note that there is a deliberate error in the XML document – this in order for us to see how to do error handling with SAX. The name of the schema file is "kompisSchema.xsd".

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="unqualified">

    <element name="kompisRelation" type="ivan:KompisRelation"/>
    <element name="person" type="ivan:Person"/>

    <attribute name="eyeColour" type="string" default="blue"/>

    <complexType name="KompisRelation">
        <sequence>
            <element name="person" type="ivan:Person"/>
            <element name="friend" type="ivan:Person" minOccurs="0"
                maxOccurs="unbounded" nillable="false"/>
        </sequence>
        <attribute name="degree" type="string" use="optional"
            default="acquaintance"/>
    </complexType>

    <complexType name="Person">
        <sequence>
            <element name="firstName" type="string" nillable="false"/>
            <element name="lastName" type="string" nillable="false"/>
            <element name="age" type="int"/>
            <element name="favColour" type="string" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
        <attribute ref="ivan:eyeColour"/>
        <attribute name="hasDog" type="boolean"/>
    </complexType>
</schema>
```

The name of the XML data file is "kompisar.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A comment in the file kompisar.xml -->
<ivan:kompisRelation
      xmlns:ivan="http://www.ivan.com/schemas"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ivan.com/schemas kompisSchema.xsd"
      degree="closeFriends">

      <person>
          <firstName>Ivan</firstName>
          <lastName>Krizsan</lastName>
          <age>19</age>
      </person>
      <friend ivan:eyeColour="green">
          <firstName>Steven</firstName>
          <lastName>Segal</lastName>
          <age>49</age>
      </friend>
      <friend>
```

```
          <firstName>Carl-Gustav</firstName>
          <lastName>Svensson</lastName>
          <age>56</age>
      </friend>

    <!-- There is a deliberate error here. -->
    <friend hasDog="true">
    </friend>
</ivan:kompisRelation>
```

Then we write an error handler, which can be used both with the SAX and with the DOM example programs.

```
package com.ivan.jaxpexamples;

import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;


/**
 * This class implements an error handler that can be used both with
 * SAX and DOM.
 *
 * @author Ivan A Krizsan
 */
public class SAXErrorHandler implements ErrorHandler
{
    /* Constant(s): */

    /* Instance variable(s): */


    /* (non-Javadoc)
     * @see org.xml.sax.ErrorHandler#error(org.xml.sax.SAXParseException)
     */
    public void error(final SAXParseException inException) throws SAXException
    {
        int theLineNo = inException.getLineNumber();
        System.out.println("*** ErrorHandler.error() : "
            + inException.getLocalizedMessage() + " at line " + theLineNo);

    }

    /* (non-Javadoc)
     * @see org.xml.sax.ErrorHandler#fatalError(org.xml.sax.SAXParseException)
     */
    public void fatalError(final SAXParseException inException) throws SAXException
    {
        int theLineNo = inException.getLineNumber();
        System.out.println("*** ErrorHandler.fatalError() : "
            + inException.getLocalizedMessage() + " at line " + theLineNo);
    }

    /* (non-Javadoc)
     * @see org.xml.sax.ErrorHandler#warning(org.xml.sax.SAXParseException)
     */
    public void warning(final SAXParseException inException) throws SAXException
    {
        int theLineNo = inException.getLineNumber();
        System.out.println("*** ErrorHandler.warning() : "
            + inException.getLocalizedMessage() + " at line " + theLineNo);
    }
}
```

SAX Processing

The following example program shows how to use SAX to count the elements in an XML document and to validate the XML document at the same time.

```
package com.ivan.jaxpexamples;

import java.io.File;
import java.util.HashMap;
import java.util.Iterator;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
import org.xml.sax.ext.DefaultHandler2;

/**
 * Example of how to use the SAX parser to parse an XML document.
 * Based on the example from the JAXP tutorial.
 *
 * @author Ivan A Krizsan
 */
public class SAXExample extends DefaultHandler2
{
    /* Constant(s): */
    private final static String KOMPIS_XML_FILE_NAME = "kompisar.xml";

    private final static String SAX_LEXICAL_HANDLER_PROPERTY =
        "http://xml.org/sax/properties/lexical-handler";
    private final static String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
    private final static String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

    /* Instance variable(s): */
    private HashMap<String, Integer> mTags;

    /**
     * @param args
     */
    public static void main(String[] args)
    {
        SAXExample theExample = new SAXExample();
        theExample.runExample();
    }

    /**
     * Runs the example program.
     */
    private void runExample()
    {
        String theFileURL;

        theFileURL = convertToFileURL(KOMPIS_XML_FILE_NAME);

        try
        {
            /* Retrieve a SAX parser factory and create a SAX parser. */
            SAXParserFactory theSAXParserFactory = SAXParserFactory.newInstance();
            theSAXParserFactory.setNamespaceAware(true);
            theSAXParserFactory.setValidating(true);
            SAXParser theSAXParser = theSAXParserFactory.newSAXParser();

            /*
             * This is required if we want the SAX parser to validate the
             * XML file. In our case, the corresponding XML schema is
             * specified in the XML file.
             */
            theSAXParser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);

            /* Get and configure the object that is to parse the XML file. */
```

```
            XMLReader theXMLReader = theSAXParser.getXMLReader();
            theXMLReader.setContentHandler(this);
            theXMLReader.setErrorHandler(new SAXErrorHandler());

            /*
             * This is needed in order to receive notifications to a
             * lexical handler.
             */
            theXMLReader.setProperty(SAX_LEXICAL_HANDLER_PROPERTY, this);

            /* Finally we are ready to parse the XML file! */
            theXMLReader.parse(theFileURL);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Converts supplied filename to an URL pointing to the file.
     *
     * @param inFileName File name.
     * @return URL pointing to file with supplied name.
     */
    private String convertToFileURL(final String inFileName)
    {
        String path = new File(inFileName).getAbsolutePath();
        if (File.separatorChar != '/')
        {
            path = path.replace(File.separatorChar, '/');
        }
        if (!path.startsWith("/"))
        {
            path = "/" + path;
        }

        return "file:" + path;
    }

    /**
     * Receives notification of start of document from the SAX parser.
     * See the org.xml.sax.ContentHandler interface for more information.
     *
     * @throws SAXException If error occurs.
     */
    @Override
    public void startDocument() throws SAXException
    {
        mTags = new HashMap<String, Integer>();

        System.out.println("*** ContentHandler.startDocument()");
    }

    /**
     * Receives notification of end of document from the SAX parser.
     * See the org.xml.sax.ContentHandler interface for more information.
     *
     * @throws SAXException If error occurs.
     */
    @Override
    public void endDocument() throws SAXException
    {
        Iterator<String> theTagsIter = mTags.keySet().iterator();
        while (theTagsIter.hasNext())
        {
            String theTag = theTagsIter.next();
            int theCount = (mTags.get(theTag)).intValue();
            System.out.println("Name \"" + theTag + "\" occurs " + theCount
                + " times");
        }

        System.out.println("*** ContentHandler.endDocument()");
    }

    /**
     * Receives notification of start of an XML element from the SAX parser.
     * See the org.xml.sax.ContentHandler interface for more information.
```

```
     *
     * @param inNamespaceURI Namespace URI of element, or empty string if
     * no URI or if namespace processing is not being done.
     * @param inLocalName Local name of element, or empty string if
     * namespace processing not done.
     * @param inQName Qualified name of element, or empty string if
     * qualified name not available.
     * @param inAttributes Attributes attached to element, or empty
     * object if no attributes.
     * @throws SAXException If error occurs.
     */
    public void startElement(final String inNamespaceURI,
        final String inLocalName, final String inQName,
        final Attributes inAttributes) throws SAXException
    {
        String theKey;

        if (!inQName.equals(""))
        {
            theKey = inQName;
        } else
        {
            theKey = inLocalName;
        }

        Integer theValue = mTags.get(theKey);
        if (theValue == null)
        {
            mTags.put(theKey, new Integer(1));
        } else
        {
            int theTagCount = theValue.intValue();
            theTagCount++;
            mTags.put(theKey, new Integer(theTagCount));
        }

        System.out.println("*** ContentHandler.startElement(" + inNamespaceURI
            + ", " + inLocalName + ", " + inQName + ")");
    }

    /**
     * Receives notifications of a comment.
     * * See the org.xml.sax.ext.LexicalHandler interface for more information.
     *
     * @param inCharacters Characters in the comment.
     * @param inStartOffset Starting offset of comment in above array.
     * @param inCharCount Number of characters of comment in above array.
     */
    @Override
    public void comment(final char[] inCharacters, final int inStartOffset,
        final int inCharCount)
    {
        System.out.println("*** LexicalHandler.comment() : "
            + new String(inCharacters) + ", start=" + inStartOffset
            + ", count=" + inCharCount);
    }
}
```

When run, the program will produce the following output:

```
*** ContentHandler.startDocument()
*** LexicalHandler.comment() :  A comment in the file kompisar.xml, start=0, count=36
*** ContentHandler.startElement(http://www.ivan.com/schemas, kompisRelation,
ivan:kompisRelation)
*** ContentHandler.startElement(, person, person)
*** ContentHandler.startElement(, firstName, firstName)
*** ContentHandler.startElement(, lastName, lastName)
*** ContentHandler.startElement(, age, age)
*** ContentHandler.startElement(, friend, friend)
*** ContentHandler.startElement(, firstName, firstName)
*** ContentHandler.startElement(, lastName, lastName)
*** ContentHandler.startElement(, age, age)
*** ContentHandler.startElement(, friend, friend)
*** ContentHandler.startElement(, firstName, firstName)
*** ContentHandler.startElement(, lastName, lastName)
*** ContentHandler.startElement(, age, age)
```

```
*** ContentHandler.startElement(, friend, friend)
*** ErrorHandler.error() : cvc-complex-type.2.4.b: The content of element 'friend' is
not complete. One of '{"":firstName}' is expected. at line 27
Name "age" occurs 3 times
Name "friend" occurs 3 times
Name "person" occurs 1 times
Name "ivan:kompisRelation" occurs 1 times
Name "firstName" occurs 3 times
Name "lastName" occurs 3 times
*** ContentHandler.endDocument()
```

By examining the XML schema, one finds that the parsing error occurs because there are required elements missing in the last <friend> declaration, the first of which is the <firstName> element. For additional comments on SAX, please refer to the section on SAX in the JAXP tutorial.

DOM Processing

In the DOM example program, we can re-use the error handler implemented in the SAX example program above. We will also use the same XML document and the associated XML schema. The class implementing the DOM example program looks like this:

```
package com.ivan.jaxpexamples;

import java.io.File;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;
import org.w3c.dom.NamedNodeMap;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

/**
 * Example of how to use the DOM parser to read and modify an XML document.
 * Based on the example from the JAXP tutorial.<br/>
 * Note that the error handler interface is the same as in the SAX
 * example, but in this example we implement it directly. The exact same
 * error handling can be used for both SAX and DOM.
 *
 * @author Ivan A Krizsan
 */
public class DOMExample
{
    /* Constant(s): */
    private final static String KOMPIS_XML_FILE_NAME = "kompisar.xml";
    private final static String JAXP_SCHEMA_LANGUAGE =
        "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
    private final static String W3C_XML_SCHEMA =
        "http://www.w3.org/2001/XMLSchema";

    /* Instance variable(s): */
    private int mIndent = 0;

    /**
     * Main entry point of program.
     *
     * @param inArguments Command-line arguments. Not used.
     */
    public static void main(String[] inArguments)
    {
        DOMExample theExample = new DOMExample();
        try
        {
            theExample.runExample();
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Runs the example program.
```

```
     *
     * @throws Exception If error occurs.
     */
    private void runExample() throws Exception
    {
        Document theKompisDocument = readXMLDocument(KOMPIS_XML_FILE_NAME);

        /* Output some information about the document. */
        System.out.println("The document : " + theKompisDocument);
        System.out.println("XML version  : "
            + theKompisDocument.getXmlVersion());
        System.out.println("XML encoding : "
            + theKompisDocument.getXmlEncoding());

        /* Output the in-memory representation of the document. */
        doTraverseNode(theKompisDocument);
    }

    /**
     * Reads and validates the XML document with the supplied name.
     *
     * @param inXMLFileName Name of XML file to read.
     * @return DOM document containing the XML document representation.
     * @throws Exception If error occurs reading XML document.
     */
    public Document readXMLDocument(final String inXMLFileName) throws Exception
    {
        DocumentBuilderFactory theDocumentBuilderFactory;
        DocumentBuilder theDocumentBuilder;
        File theXMLFile;
        Document theKompisDocument;

        /*
         * Retrieve the factory which creates instances of objects that
         * read XML files.
         */
        theDocumentBuilderFactory = DocumentBuilderFactory.newInstance();
        /*
         * Configure so that documents read are validated and so that the
         * parser understands namespaces.
         */
        theDocumentBuilderFactory.setValidating(true);
        theDocumentBuilderFactory.setNamespaceAware(true);
        /*
         * Configure parser to ignore whitespace in element content.
         * When using this feature, the parser must also be set to
         * validate.
         */
        theDocumentBuilderFactory.setIgnoringElementContentWhitespace(true);
        /*
         * Set the schema language to XML, in order for DOM to be able to
         * validate our document.
         * The XML schema used to validate our document is, in this case,
         * specified in the XML document.
         */
        theDocumentBuilderFactory.setAttribute(JAXP_SCHEMA_LANGUAGE,
            W3C_XML_SCHEMA);

        /*
         * Retrieve a document builder that reads an XML file and
         * set its error handler to this object.
         */
        theDocumentBuilder = theDocumentBuilderFactory.newDocumentBuilder();
        theDocumentBuilder.setErrorHandler(new SAXErrorHandler());

        /* Read the XML file. */
        theXMLFile = new File(inXMLFileName);
        theKompisDocument = theDocumentBuilder.parse(theXMLFile);
        return theKompisDocument;
    }

    /**
     * Outputs information about the supplied DOM node and all of its children.
     *
     * @param inNode Node for which to output information.
     */
    private void doTraverseNode(final Node inNode)
```

```
    {
        outputNodeInformation(inNode);

        mIndent += 4;

        NodeList theChildNodes = inNode.getChildNodes();
        for (int i = 0; i < theChildNodes.getLength(); i++)
        {
            Node theChildNode = theChildNodes.item(i);
            doTraverseNode(theChildNode);
        }

        mIndent -= 4;

        outputIndentation();
        System.out.println("End of node=\"" + inNode.getNodeName() + "\"");
    }

    /**
     * Outputs information about the supplied DOM node.
     *
     * @param inNode Node for which to output information.
     */
    private void outputNodeInformation(final Node inNode)
    {
        outputIndentation();

        System.out.print("NodeName=\"" + inNode.getNodeName() + "\"");

        String theStr = inNode.getNamespaceURI();
        printNonNull(", uri=", theStr);

        theStr = inNode.getPrefix();
        printNonNull(", pre=", theStr);

        theStr = inNode.getLocalName();
        printNonNull(", local=", theStr);

        theStr = inNode.getNodeValue();
        if (theStr != null)
        {
            System.out.print(", nodeValue=");
            if (theStr.trim().equals(""))
            {
                System.out.print("[empty]");
            } else
            {
                System.out.print("\"" + inNode.getNodeValue() + "\"");
            }
        }

        /* Output the attributes of the node, if any. */
        NamedNodeMap theAttributes = inNode.getAttributes();
        if (theAttributes != null)
        {
            int theAttributesCount = theAttributes.getLength();
            if (theAttributesCount > 0)
            {
                System.out.print(", attributes: ");
                for(int i = 0; i < theAttributesCount; i ++)
                {
                    Node theAttribute = theAttributes.item(i);
                    System.out.print(theAttribute.getNodeName() + "=" +
                        theAttribute.getNodeValue() + ", ");
                }
            }
        }

        System.out.println();
    }

    /**
     * Prints the supplied name and value, if the value is not null.
     * The value will be printed in quotes.
     *
     * @param inName Name to print.
     * @param inValue Value to print, or null.
```

```
     */
    private void printNonNull(final String inName, final String inValue)
    {
        if (inValue != null)
        {
            System.out.print(inName + "\"" + inValue + "\"");
        }
    }

    private void outputIndentation()
    {
        for (int i = 0; i < mIndent; i++)
        {
            System.out.print(" ");
        }
    }
}
```

When run, the program produces the following output:

```
*** ErrorHandler.error() : cvc-complex-type.2.4.b: The content of element 'friend' is
not complete. One of '{"":firstName}' is expected. at line 27
The document : [#document: null]
XML version  : 1.0
XML encoding : UTF-8
NodeName="#document"
    NodeName="#comment", nodeValue=" A comment in the file kompisar.xml "
    End of node="#comment"
    NodeName="ivan:kompisRelation", uri="http://www.ivan.com/schemas", pre="ivan",
local="kompisRelation", attributes: degree=closeFriends, xmlns:ivan=http://ww
w.ivan.com/schemas, xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance,
xsi:schemaLocation=http://www.ivan.com/schemas kompisSchema.xsd,
        NodeName="#text", nodeValue=[empty]
        End of node="#text"
        NodeName="person", local="person", attributes: eyeColour=blue,
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="firstName", local="firstName"
                NodeName="#text", nodeValue="Ivan"
                End of node="#text"
            End of node="firstName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="lastName", local="lastName"
                NodeName="#text", nodeValue="Krizsan"
                End of node="#text"
            End of node="lastName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="age", local="age"
                NodeName="#text", nodeValue="19"
                End of node="#text"
            End of node="age"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
        End of node="person"
        NodeName="#text", nodeValue=[empty]
        End of node="#text"
        NodeName="friend", local="friend", attributes: ivan:eyeColour=green,
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="firstName", local="firstName"
                NodeName="#text", nodeValue="Steven"
                End of node="#text"
            End of node="firstName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="lastName", local="lastName"
                NodeName="#text", nodeValue="Segal"
                End of node="#text"
            End of node="lastName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="age", local="age"
                NodeName="#text", nodeValue="49"
                End of node="#text"
```

```
            End of node="age"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
        End of node="friend"
        NodeName="#text", nodeValue=[empty]
        End of node="#text"
        NodeName="friend", local="friend", attributes: eyeColour=blue,
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="firstName", local="firstName"
                NodeName="#text", nodeValue="Carl-Gustav"
                End of node="#text"
            End of node="firstName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="lastName", local="lastName"
                NodeName="#text", nodeValue="Svensson"
                End of node="#text"
            End of node="lastName"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
            NodeName="age", local="age"
                NodeName="#text", nodeValue="56"
                End of node="#text"
            End of node="age"
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
        End of node="friend"
        NodeName="#text", nodeValue=[empty]
        End of node="#text"
        NodeName="friend", local="friend", attributes: eyeColour=blue, hasDog=true,
            NodeName="#text", nodeValue=[empty]
            End of node="#text"
        End of node="friend"
        NodeName="#text", nodeValue=[empty]
        End of node="#text"
    End of node="ivan:kompisRelation"
End of node="#document"
```

The following things should be noticed:

- The global attribute eyeC*olour* which has been given the default value “blue” appears in all instances of the *Person* complex type.
- Despite the parsing error, the document, including the faulty element, is still read.

For additional comments on DOM, please refer to the section on DOM in the JAXP tutorial.

StAX Processing

The following example program will show how to use StAX to:

- Read XML data using the iterator API.
- Write XML data using the iterator API.
- Read XML data using the cursor API.
- Write XML data using the cursor API.

The first XML data file, named “kompisar.xml”, is the same as in listed in the prerequisites.
The second XML data file is identical to kompisar.xml, except that the incomplete element has been removed. It is called “kompisar_noerrors.xml”.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A comment in the file kompisar_noerrors.xml -->
<ivan:kompisRelation
      xmlns:ivan="http://www.ivan.com/schemas"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ivan.com/schemas kompisSchema.xsd"
      degree="closeFriends">

      <person>
            <firstName>Ivan</firstName>
            <lastName>Krizsan</lastName>
            <age>19</age>
      </person>
      <friend ivan:eyeColour="green">
         <firstName>Steven</firstName>
         <lastName>Segal</lastName>
         <age>49</age>
      </friend>
      <friend>
         <firstName>Carl-Gustav</firstName>
         <lastName>Svensson</lastName>
         <age>56</age>
      </friend>
</ivan:kompisRelation>
```

The XML schema file referred to by the above files was also listed in the prerequisites.

The example code is implemented in one single class. In order to run the different example programs, the static *main* method should be modified.

```
package com.ivan.jaxpexamples;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.util.Iterator;

import javax.xml.namespace.QName;
import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import javax.xml.stream.XMLStreamWriter;
import javax.xml.stream.events.Attribute;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.XMLEvent;

/**
 * Example of how to use the StAX parser to read, modify and write
 * an XML document. Based on the example from the JAXP tutorial.<br/>
 * This example program requires JavaSE 6.
 *
 * @author Ivan A Krizsan
 */
public class StAXExample
{
    /* Constant(s): */
    private final static String KOMPIS_XML_FILE_NAME = "kompisar.xml";
    private final static String KOMPIS_NOERR_XML_FILE_NAME =
        "kompisar_noerrors.xml";
    private final static String ONE_TAB = "    ";
    private final static String XML_ITERATOR_OUTPUT_FILE =
        "stax_iterator_output.xml";
    private final static String XML_CURSOR_OUTPUT_FILE =
        "stax_cursor_output.xml";

    /* Instance variable(s): */

    /**
     * Program main entry point.
     *
     * @param args Command line arguments, ignored.
     */
    public static void main(final String[] args)
    {
        final StAXExample theExample = new StAXExample();
        try
        {
            theExample.readWriteWithCursor();
        } catch (final Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private void printTab(final int inTabs)
    {
        for (int i = 0; i < inTabs; i++)
        {
            System.out.print(ONE_TAB);
        }
    }

    /**
     * Reads an XML file using the StAX iterator API and prints data for
     * some of the event types.
     *
     * @throws Exception If error occurs reading file.
     */
```

```
    public void readWithIterator() throws Exception
    {
        int theTab = 0;

        System.out
            .println("***** Reading XML file using the StAX Iterator API:");

        /*
         * Retrieve a factory that can create both StAX event and
         * stream readers.
         */
        final XMLInputFactory theFactory = XMLInputFactory.newInstance();

        /*
         * Create an event reader, also called an iterator, for reading
         * the example XML document.
         */
        final FileInputStream theFileIS =
            new FileInputStream(KOMPIS_XML_FILE_NAME);
        final XMLEventReader theXMLIterator =
            theFactory.createXMLEventReader(theFileIS);

        while (theXMLIterator.hasNext())
        {
            final XMLEvent theEvent = theXMLIterator.nextEvent();

            switch (theEvent.getEventType())
            {
                case XMLStreamConstants.CHARACTERS:
                    final String theCharacters =
                        theEvent.asCharacters().toString().trim();
                    if (theCharacters.length() > 0)
                    {
                        printTab(theTab);
                        System.out
                            .println("Characters event: " + theCharacters);
                    }
                    break;
                case XMLStreamConstants.COMMENT:
                    printTab(theTab);
                    System.out.println("Comment event: " + theEvent.toString());
                    break;
                case XMLStreamConstants.END_DOCUMENT:
                    printTab(theTab);
                    System.out.println("End of document event.");
                    break;
                case XMLStreamConstants.END_ELEMENT:
                    printTab(--theTab);
                    final EndElement theEndElement = theEvent.asEndElement();
                    System.out.println("End of element: "
                        + theEndElement.getName());
                    break;
                case XMLStreamConstants.NAMESPACE:
                    printTab(theTab);
                    System.out.println("Namespace event: "
                        + theEvent.asCharacters());
                    break;
                case XMLStreamConstants.START_DOCUMENT:
                    printTab(theTab);
                    System.out.println("Start of document");
                    break;
                case XMLStreamConstants.START_ELEMENT:
                    printTab(theTab++);
                    final StartElement theStartElement =
                        theEvent.asStartElement();
                    System.out.print("Start of element: "
                        + theStartElement.getName());

                    /* List all the attributes, if any. */
                    final Iterator<Attribute> theAttrIter =
                        theStartElement.getAttributes();
                    if (theAttrIter.hasNext())
                    {
                        System.out.print(", attributes: ");
                    }
                    while (theAttrIter.hasNext())
                    {
```

```
                    final Attribute theAttribute = theAttrIter.next();
                    System.out.print(theAttribute.getName() + "="
                        + theAttribute.getValue());
                    if (theAttrIter.hasNext())
                    {
                        System.out.print(", ");
                    }
                }
                System.out.println("");
                break;
            default:
                printTab(theTab);
                System.out.println("Unrecognized event: "
                    + theEvent.getEventType());
        }
    }
}

/**
 * Reads an XML file using the StAX stream API and prints data for
 * some of the event types.
 *
 * @throws Exception If error occurs reading file.
 */
public void readWithCursor() throws Exception
{
    int theTab = 0;

    System.out.println("***** Reading XML file using the StAX Cursor API:");

    /*
     * Retrieve a factory that can create both StAX event and
     * stream readers.
     */
    final XMLInputFactory theFactory = XMLInputFactory.newInstance();

    /*
     * Create a stream reader, also called an cursor, for reading
     * the example XML document.
     */
    final FileInputStream theFileIS =
        new FileInputStream(KOMPIS_XML_FILE_NAME);
    final XMLStreamReader theXMLCursor =
        theFactory.createXMLStreamReader(theFileIS);

    while (theXMLCursor.hasNext())
    {
        final int theEventCode = theXMLCursor.next();

        switch (theEventCode)
        {
            case XMLStreamConstants.CHARACTERS:
                final String theCharacters = theXMLCursor.getText().trim();
                if (theCharacters.length() > 0)
                {
                    printTab(theTab);
                    System.out
                        .println("Characters event: " + theCharacters);
                }
                break;
            case XMLStreamConstants.COMMENT:
                printTab(theTab);
                System.out.println("Comment event: "
                    + theXMLCursor.getText());
                break;
            case XMLStreamConstants.END_DOCUMENT:
                printTab(theTab);
                System.out.println("End of document event.");
                break;
            case XMLStreamConstants.END_ELEMENT:
                printTab(--theTab);
                System.out.println("End of element: "
                    + theXMLCursor.getLocalName());
                break;
            case XMLStreamConstants.NAMESPACE:
                printTab(theTab);
                System.out.println("Namespace event: ");
```

```
                break;
            case XMLStreamConstants.START_DOCUMENT:
                printTab(theTab);
                System.out.println("Start of document");
                break;
            case XMLStreamConstants.START_ELEMENT:
                printTab(theTab++);
                System.out.print("Start of element: "
                    + theXMLCursor.getLocalName());

                /* List all the attributes, if any. */
                final int theAttributeCount =
                    theXMLCursor.getAttributeCount();
                if (theAttributeCount > 0)
                {
                    System.out.print(", attributes: ");
                }
                for (int i = 0; i < theAttributeCount; i++)
                {
                    System.out.print(theXMLCursor.getAttributeLocalName(i)
                        + "=" + theXMLCursor.getAttributeValue(i));
                    if ((i + 1) < theAttributeCount)
                    {
                        System.out.print(", ");
                    }
                }
                System.out.println();
                break;
            default:
                printTab(theTab);
                System.out.println("Unrecognized event: " + theEventCode);
        }
    }
}

/**
 * Reads an XML file using the StAX iterator API and and writes it,
 * also using the StAX iterator API, to an output file.
 * Some additional processing will be done before the data is written.
 *
 * @throws Exception If error occurs reading file.
 */
public void readWriteWithIterator() throws Exception
{
    System.out
        .println("***** Read and write XML using the StAX Iterator API:");

    /*
     * Retrieve factories that can create both StAX event and
     * stream readers and writers.
     */
    final XMLInputFactory theInputFactory = XMLInputFactory.newInstance();
    final XMLOutputFactory theOutputFactory =
        XMLOutputFactory.newInstance();

    /* Create a factory that can create XML events. */
    final XMLEventFactory theEventFactory = XMLEventFactory.newInstance();

    /*
     * Create an event reader, also called an iterator, for reading
     * the example XML document.
     */
    final FileInputStream theFileIS =
        new FileInputStream(KOMPIS_XML_FILE_NAME);
    final XMLEventReader theXMLIterator =
        theInputFactory.createXMLEventReader(theFileIS);

    /* Create an event writer in order to write to a file. */
    final FileOutputStream theFOS =
        new FileOutputStream(XML_ITERATOR_OUTPUT_FILE);
    final XMLEventWriter theXMLWriter =
        theOutputFactory.createXMLEventWriter(theFOS);

    while (theXMLIterator.hasNext())
    {
        final XMLEvent theEvent = theXMLIterator.nextEvent();
```

```
            switch (theEvent.getEventType())
            {
                case XMLStreamConstants.CHARACTERS:
                {
                    /* Change all character data to uppercase only. */
                    XMLEvent theNewEvent;
                    String theOriginalCharacters;

                    theOriginalCharacters =
                        theEvent.asCharacters().getData().trim();
                    if (theOriginalCharacters.length() > 0)
                    {
                        theNewEvent =
                            theEventFactory
                                .createCharacters(theOriginalCharacters
                                    .toUpperCase());
                        theXMLWriter.add(theNewEvent);
                    }
                    break;
                }
                case XMLStreamConstants.COMMENT:
                {
                    /* Reverse the text of all comments. */
                    XMLEvent theNewEvent;
                    StringBuffer theComment;

                    theComment = new StringBuffer(theEvent.toString());
                    theComment.reverse();

                    theNewEvent =
                        theEventFactory.createComment(theComment.toString());

                    theXMLWriter.add(theNewEvent);
                    break;
                }
                case XMLStreamConstants.END_DOCUMENT:
                    theXMLWriter.add(theEvent);
                    break;
                case XMLStreamConstants.END_ELEMENT:
                    theXMLWriter.add(theEvent);
                    break;
                case XMLStreamConstants.NAMESPACE:
                    System.err.println("A namespace event was ignored");
                    break;
                case XMLStreamConstants.START_DOCUMENT:
                    theXMLWriter.add(theEvent);
                    break;
                case XMLStreamConstants.START_ELEMENT:
                    Attribute theNewAttribute;

                    theXMLWriter.add(theEvent);

                    /* Add a new attribute to all elements. */
                    theNewAttribute =
                        theEventFactory.createAttribute("newAttribute",
                            "someValue");
                    theXMLWriter.add(theNewAttribute);

                    break;
                default:
                    System.err.println("Unrecognized event was ignored: "
                        + theEvent.getEventType());
                    break;
            }
        }
        theFOS.close();
    }

    /**
     * Reads an XML file using the StAX stream API and and writes it,
     * also using the StAX stream API, to an output file.
     * Some additional processing will be done before the data is written.
     *
     * @throws Exception If error occurs reading file.
     */
    public void readWriteWithCursor() throws Exception
    {
```

```
        System.out
            .println("***** Reading and writing XML file using the StAX Cursor API:");

        (new File(XML_CURSOR_OUTPUT_FILE)).delete();

        /*
         * Retrieve factories that can create both StAX event and
         * stream readers and writers.
         * If we set the IS_REPAIRING_NAMESPACES property to true, the
         * correct namespace declarations will automatically be present
         * in the output file.
         */
        final XMLInputFactory theInputFactory = XMLInputFactory.newInstance();
        final XMLOutputFactory theOutputFactory =
            XMLOutputFactory.newInstance();
        theOutputFactory.setProperty(XMLOutputFactory.IS_REPAIRING_NAMESPACES,
            Boolean.TRUE);

        /*
         * Create a stream reader, also called an cursor, for reading
         * the example XML document.
         */
        final FileInputStream theFileIS =
            new FileInputStream(KOMPIS_NOERR_XML_FILE_NAME);
        final XMLStreamReader theXMLCursor =
            theInputFactory.createXMLStreamReader(theFileIS);

        /* Create an stream writer in order to write to a file. */
        final FileWriter theOutputFileWriter =
            new FileWriter(XML_CURSOR_OUTPUT_FILE);
        final XMLStreamWriter theXMLWriter =
            theOutputFactory.createXMLStreamWriter(theOutputFileWriter);

        /*
         * We will not receive a START_DOCUMENT event, so we have to
         * put the code that initializes the output file here.
         */
        theXMLWriter.writeStartDocument("UTF-8", "1.0");

        while (theXMLCursor.hasNext())
        {
            final int theEventCode = theXMLCursor.next();

            switch (theEventCode)
            {
                case XMLStreamConstants.CHARACTERS:
                {
                    System.out.println("*** Characters");

                    /* Change all character data to lowercase only. */
                    final String theCharacters = theXMLCursor.getText().trim();
                    if (theCharacters.length() > 0)
                    {
                        theXMLWriter.writeCharacters(theCharacters
                            .toLowerCase());
                    }
                    break;
                }
                case XMLStreamConstants.COMMENT:
                {
                    System.out.println("*** Comment");

                    /* Change all comments to uppercase only. */
                    final String theComment =
                        theXMLCursor.getText().toUpperCase();
                    theXMLWriter.writeComment(theComment);
                    break;
                }
                case XMLStreamConstants.END_DOCUMENT:
                    System.out.println("*** End document");

                    theXMLWriter.writeEndDocument();
                    theXMLWriter.flush();
                    theXMLWriter.close();
                    break;
                case XMLStreamConstants.END_ELEMENT:
                    System.out.println("*** End element");
```

```
                    theXMLWriter.writeEndElement();
                    break;
                case XMLStreamConstants.NAMESPACE:
                    System.err.println("--- A namespace event was ignored");
                    break;
                case XMLStreamConstants.START_DOCUMENT:
                    System.out.println("--- Start document");
                    break;
                case XMLStreamConstants.START_ELEMENT:
                    System.out.println("*** Start element");

                    /*
                     * If the name originally is qualified, then we want
                     * to output a qualified name, otherwise output a
                     * local name only.
                     */
                    final QName theElementQName = theXMLCursor.getName();
                    final String theNameSpace =
                        theElementQName.getNamespaceURI();
                    if (theNameSpace.length() > 0)
                    {
                        theXMLWriter.writeStartElement(theElementQName
                            .getPrefix(), theElementQName.getLocalPart(),
                            theNameSpace);
                    } else
                    {
                        theXMLWriter.writeStartElement(theElementQName
                            .getLocalPart());
                    }

                    /* List all the attributes, if any. */
                    final int theAttributeCount =
                        theXMLCursor.getAttributeCount();
                    for (int i = 0; i < theAttributeCount; i++)
                    {
                        final QName theAttributeQName =
                            theXMLCursor.getAttributeName(i);
                        final String theAttributeValue =
                            theXMLCursor.getAttributeValue(i);

                        /*
                         * If the attribute has a namespace specified, then
                         * write the fully qualified name of the attribute,
                         * otherwise just write the local name.
                         */
                        final String theAttributeNameSpace =
                            theAttributeQName.getNamespaceURI();
                        if (theAttributeNameSpace.length() > 0)
                        {
                            theXMLWriter.writeAttribute(theAttributeQName
                                .getPrefix(), theAttributeNameSpace,
                                theAttributeQName.getLocalPart(),
                                theAttributeValue);
                        } else
                        {
                            theXMLWriter.writeAttribute(theAttributeQName
                                .getLocalPart(), theAttributeValue);
                        }
                    }
                    break;

                default:
                    System.err.println("Unrecognized event: " + theEventCode);
            }
        }
    }
}
```

XSLT Processing

The following example program will show how to use XSLT to:

- Write an entire DOM document to an XML file.
  This is basically a transformation that does not change the data at all.
- Write a part of a DOM document to an XML file.
- Transform data, in this example key-value data, to an XML document.
- Transform XML to data, in this example HTML.

The example program will use part of the DOM example program to read XML files.
Note that the examples focus on how to use the XSLT API and does not attempt to teach XSL, the extensible stylesheet language used to define transforms.
The XML data file, name "kompisar_noerrors.xml", is the same as in the StAX example program.
The XML schema file referred to by the data file is the same as listed in the prerequisites.
Next there is a key-value data file that will be transformed to XML. Its name is "key_value_data.txt".

```
name: Steven Stegal
address: 1012 Ohoio Street
state: Illinois
zipcode: 60612
phone: 555-123-1234
```

For the XML to HTML transformation example we also need a XSL file. No further explanations will be given for this file, since it is outside the scope of this document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:ivan="http://www.ivan.com/schemas">

    <xsl:output method="html" />

    <xsl:template match="/">
        <html>
            <body>
                <xsl:apply-templates />
            </body>
        </html>
    </xsl:template>

    <xsl:template match="/ivan:kompisRelation">
        <h1 align="center">
            <xsl:apply-templates />
        </h1>
    </xsl:template>

    <!-- Print the person who has the friends -->
    <xsl:template match="/ivan:kompisRelation/person">
        <h2 align="center">
            Friends of
            <xsl:apply-templates select="firstName" />
        </h2>
    </xsl:template>

    <!-- List the friends -->
    <xsl:template match="/ivan:kompisRelation/friend">
        <li>
            <xsl:apply-templates select="firstName" />
        </li>
    </xsl:template>

</xsl:stylesheet>
```

The example code is implemented in one single class, XSLTExample.java:

```
package com.ivan.jaxpexamples;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.sax.SAXSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.xml.sax.InputSource;

/**
 * Example how to use XSLT to do the following things:<br/>
 * - Writing a DOM to an XML file.<br/>
 * - Write the part of a DOM-tree to an XML file.<br/>
 * - Transform a data file, key-value in this case, to an XML file.</br>
 * - Transform an XML file to a HTML file, using a transform document.<br/>
 *
 * @author Ivan A Krizsan
 */
public class XSLTExample
{
    /* Constant(s): */
    private final static String KOMPIS_XML_FILE_NAME = "kompisar_noerrors.xml";
    private final static String PERSON_ELEMENT_NAME = "person";
    private final static String KEY_VALUE_FILE_NAME = "key_value_data.txt";
    private final static String STYLESHEET_FILE_NAME = "xml2html.xsl";
    private final static String  HTML_FILE_NAME = "output.html";

    /* Instance variable(s): */

    /**
     * Program main entry point.
     *
     * @param args Command line arguments. Not used.
     */
    public static void main(String[] args)
    {
        XSLTExample theInstance = new XSLTExample();
        try
        {
            theInstance.doTransformXML2HTML();
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }

    }

    /**
     * Reads an XML document into a DOM and using XSLT, writes the
     * entire DOM to the console unchanged.
     *
     * @throws Exception If error occurs reading XML document.
     */
    public void doWriteEntireXMLExample() throws Exception
    {
        DOMExample theDOMReader = new DOMExample();
        Document theDOM;

        /* Use DOM to read the document. */
        theDOM = theDOMReader.readXMLDocument(KOMPIS_XML_FILE_NAME);

        /* Get the XSLT transformer factory and create a new transformer. */
        TransformerFactory theXFormFactory = TransformerFactory.newInstance();
        Transformer theXFormer = theXFormFactory.newTransformer();

        /* Configure transformer to produce indented output. */
```

```
        theXFormer.setOutputProperty(OutputKeys.INDENT, "yes");

        /* Create a source and result for the transformer. */
        DOMSource theDOMSrc = new DOMSource(theDOM);
        StreamResult theResult = new StreamResult(System.out);

        /* Do the transform, which just passes data through. */
        theXFormer.transform(theDOMSrc, theResult);
    }

    /**
     * Reads an XML document into a DOM and using XSLT, writes a
     * part of the DOM to the console.
     *
     * @throws Exception If error occurs reading XML document.
     */
    public void doWritePartXMLExample() throws Exception
    {
        DOMExample theDOMReader = new DOMExample();
        Document theDOM;

        /* Use DOM to read the document. */
        theDOM = theDOMReader.readXMLDocument(KOMPIS_XML_FILE_NAME);

        /* Get the XSLT transformer factory and create a new transformer. */
        TransformerFactory theXFormFactory = TransformerFactory.newInstance();
        Transformer theXFormer = theXFormFactory.newTransformer();

        /*
         * Configure transformer to produce:
         * - Indented output.
         * - A standalone document.
         */
        theXFormer.setOutputProperty(OutputKeys.INDENT, "yes");
        theXFormer.setOutputProperty(OutputKeys.STANDALONE, "yes");

        /*
         * Create a source and result for the transformer.
         * Note that a node in the DOM is selected to be the root
         * of the XML document to be output.
         */
        Node thePersonNode =
            theDOM.getElementsByTagName(PERSON_ELEMENT_NAME).item(0);
        DOMSource theDOMSrc = new DOMSource(thePersonNode);
        StreamResult theResult = new StreamResult(System.out);

        /* Do the transform. */
        theXFormer.transform(theDOMSrc, theResult);
    }

    /**
     * Transforms a key-value file to XML using XSLT transformation with
     * a custom-written parser.
     *
     * @see KeyValueFileReader
     * @throws Exception If error occurs reading file.
     */
    public void doTransformKeyPropFile2XML() throws Exception
    {
        /* Specify from which file to read key-value data. */
        FileReader theKeyValueFileReader = new FileReader(KEY_VALUE_FILE_NAME);

        /* Buffer the data read from the key-value file. */
        BufferedReader theKeyValueBufReader =
            new BufferedReader(theKeyValueFileReader);
        /*
         * Create an input source for the key-value file.
         * The input source is later used by a SAXSource object to
         * read the data of the key-value file.
         */
        InputSource theKeyValueInputSource =
            new InputSource(theKeyValueBufReader);

        /*
         * Create a source that reads data from the input source created
         * above and parses it using our key-value file parser.
         */
```

```
        KeyValueFileReader theKeyValueFileParser = new KeyValueFileReader();
        SAXSource theSAXSource =
            new SAXSource(theKeyValueFileParser, theKeyValueInputSource);

        /* Get the XSLT transformer factory and create a new transformer. */
        TransformerFactory theXFormFactory = TransformerFactory.newInstance();
        Transformer theXFormer = theXFormFactory.newTransformer();

        /* Configure transformer to produce indented output. */
        theXFormer.setOutputProperty(OutputKeys.INDENT, "yes");

        /*
         * Transformer is to output the result of the transformation to
         * the console.
         */
        StreamResult theResult = new StreamResult(System.out);

        /* Do the transform. */
        theXFormer.transform(theSAXSource, theResult);
    }

    /**
     * Transforms an XML file to a HTML document.
     *
     * @throws Exception If error occurs.
     */
    public void doTransformXML2HTML() throws Exception
    {
        DOMExample theDOMReader = new DOMExample();
        Document theDOM;

        /* Erase any previous output file. */
        File theOutputFile = new File(HTML_FILE_NAME);
        theOutputFile.delete();

        /* XSLT stylesheet to be used when transforming. */
        File theStylesheetFile = new File(STYLESHEET_FILE_NAME);
        StreamSource theStylesheetSource = new StreamSource(theStylesheetFile);

        /* Use DOM to read the document. */
        theDOM = theDOMReader.readXMLDocument(KOMPIS_XML_FILE_NAME);

        /*
         * Get the XSLT transformer factory and create a new transformer.
         * Note that the stylesheet source is used when getting a new
         * transformer from the factory.
         */
        TransformerFactory theXFormFactory = TransformerFactory.newInstance();
        Transformer theXFormer = theXFormFactory.newTransformer(theStylesheetSource);

        /* Configure transformer to produce indented output. */
        theXFormer.setOutputProperty(OutputKeys.INDENT, "yes");

        /* Create a source and result for the transformer. */
        DOMSource theDOMSrc = new DOMSource(theDOM);
        StreamResult theResult = new StreamResult(theOutputFile);

        /* Do the transform. */
        theXFormer.transform(theDOMSrc, theResult);

        System.out.println("Finished writing HTML file...");
    }
}
```

JAXB Processing

There will be two example programs for JAXB:

- XML to Java
  Read an XML document and obtain a corresponding object hierarchy.
- Java to XML
  Start with a Java object hierarchy and generate an XML document.

XML to Java

If the XML schema from the Prerequisites section is bound, the following classes will be produced:

- Person.java
  Derived from the XML complex type Person.
- KompisRelation.java
  Derived from the XML complex type KompisRelation.
- ObjectFactory.java
  Factory class that allows for programmatic creation of new instances of Java representation for XML content.
- package-info.java
  This is, strictly speaking, not a class but only holds a package-level annotation.

The XML schema was bound using the XJC Plugin, which is available for Eclipse and IntelliJ IDEs. This plugin aids in creating JAXB bean classes from XML schema. Please download and install this plugin, if you haven't already.
The following code is then used to unmarshal the XML document from the Prerequisites section:

```
    /* Constant(s): */
    private final static String KOMPISAR_XML_FILE_NAME = "kompisar_noerrors.xml";
    private final static String KOMPISAR_XML_SCHEMA_FILE_NAME = "kompisSchema.xsd";

    /**
     * Unmarshals an XML document, creating an object tree representing
     * the contents of the XML document.
     * Then outputs data to the console.
     *
     * @throws Exception If error occurs unmarshaling the XML document.
     */
    public void unmarshallXMLDocument() throws Exception
    {
        /*
         * Create a JAXB context whose context path is com.ivan.beans,
         * the package in which the classes generated from the XML
         * schema are contained.
         */
        JAXBContext theJAXBContext = JAXBContext.newInstance("com.ivan.beans");

        /* Create the object responsible for unmarshaling XML document(s). */
        Unmarshaller theUnmarshaller = theJAXBContext.createUnmarshaller();

        /*
         * Validate the XML document against its schema, as part of
         * the unmarshaling operation.
         * Having set a schema for the unmarshaller, an exception will
         * be thrown if the XML document does not pass validation.
         */
        SchemaFactory theSchemaFactory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema theKompisSchema =
            theSchemaFactory.newSchema(new File(KOMPISAR_XML_SCHEMA_FILE_NAME));
        theUnmarshaller.setSchema(theKompisSchema);

        /* Unmarshal the XML document. */
```

```
        JAXBElement<KompisRelation> theJAXBKR =
            (JAXBElement<KompisRelation>)theUnmarshaller.unmarshal(new File(
                KOMPISAR_XML_FILE_NAME));

        /* Retrieve the content model for the JAXB element. */
        KompisRelation theKR = theJAXBKR.getValue();

        /*
         * Output the friend-relation described by the XML document,
         * now represented by the KompisRelation object tree.
         */
        Person thePerson = theKR.getPerson();
        System.out.println(thePerson.getFirstName() + " "
            + thePerson.getLastName() + " has the following friends:");

        List<Person> theFriends = theKR.getFriend();
        for (Person theFriend : theFriends)
        {
            System.out.println(" - " + theFriend.getFirstName() + " "
                + theFriend.getLastName());
        }
    }
```

Not surprisingly, the above program generates the following console output:

```
Ivan Krizsan has the following friends:
 - Steven Segal
 - Carl-Gustav Svensson
```

Java-to-XML

Since we already have bound the XML schema, there is no need to redo it - we'll just use the previously created classes. The following code snippet contains a method that creates the content tree and another method that marshals the content tree to an XML document.

```
    private final static String KOMPISAR_XML_SCHEMA_FILE_NAME = "kompisSchema.xsd";
    private final static String KOMPISAR_XML_SCHEMA_NAMESPACE =
        "http://www.ivan.com/schemas";
    private final static String KOMPISAR_XML_SCHEMALOCATION =
        KOMPISAR_XML_SCHEMA_NAMESPACE + " " + KOMPISAR_XML_SCHEMA_FILE_NAME;

    /**
     * Marshals an object tree to an XML document and writes the
     * result to a file.
     * The produced XML document is validated against appropriate
     * XML schema, as it is produced.
     *
     * @throws Exception If error occurs marshaling the document.
     */
    public void marshalXMLDocument() throws Exception
    {
        JAXBElement<KompisRelation> theRelationWrapper;

        /* Creates the object tree describing the friends of a person. */
        theRelationWrapper = setupFriendRelation();

        /*
         * Create a JAXB context whose context path is com.ivan.beans,
         * the package in which the classes generated from the XML
         * schema are contained.
         */
        JAXBContext theJAXBContext = JAXBContext.newInstance("com.ivan.beans");

        /*
         * Create the object responsible for marshaling XML document(s).
         * Set a property that causes the output to be formatted so that
         * it is easier to read.
         */
        Marshaller theMarshaller = theJAXBContext.createMarshaller();
        theMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
            Boolean.TRUE);

        /*
         * Set a property specifying the value of the xsi:schemaLocation
         * property written to the XML document.
```

```
         */
        theMarshaller.setProperty(Marshaller.JAXB_SCHEMA_LOCATION,
            KOMPISAR_XML_SCHEMALOCATION);

        /*
         * Specifying a schema for the marshaller causes it to validate
         * the XML document produced when marshaling.
         */
        SchemaFactory theSchemaFactory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema theKompisSchema =
            theSchemaFactory.newSchema(new File(KOMPISAR_XML_SCHEMA_FILE_NAME));
        theMarshaller.setSchema(theKompisSchema);

        /* Create a writer that writes the XML document to the console. */
        PrintWriter theWriter = new PrintWriter(System.out);

        theMarshaller.marshal(theRelationWrapper, theWriter);

        theWriter.close();
    }

    private JAXBElement<KompisRelation> setupFriendRelation()
    {
        ObjectFactory theFactory;
        Person thePerson;
        Person theFriend;
        KompisRelation theRelation;

        /*
         * This is the factory creating the Java objects of the content
         * tree to be marshalled into an XML document.
         */
        theFactory = new ObjectFactory();

        /*
         * First an object tree needs to be created and data need to be
         * inserted.
         * Set up the data about the person which friends are to be listed.
         */
        thePerson = theFactory.createPerson();
        thePerson.setAge(28);
        thePerson.setEyeColour("brown");
        thePerson.setFirstName("Izzy");
        thePerson.setLastName("Stradlin");
        thePerson.setHasDog(true);

        /* Set up the friends of the above person. */
        theRelation = theFactory.createKompisRelation();
        theRelation.setPerson(thePerson);

        theFriend = theFactory.createPerson();
        theFriend.setAge(29);
        theFriend.setEyeColour("blue");
        theFriend.setFirstName("Michael");
        theFriend.setLastName("Monroe");
        theFriend.setHasDog(false);

        theRelation.getFriend().add(theFriend);

        theFriend = theFactory.createPerson();
        theFriend.setAge(24);
        theFriend.setEyeColour("green");
        theFriend.setFirstName("Razzle");
        theFriend.setLastName("");
        theFriend.setHasDog(false);

        theRelation.getFriend().add(theFriend);

        return theFactory.createKompisRelation(theRelation);
    }
```

Running the above program produces the following console output:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:kompisRelation xsi:schemaLocation="http://www.ivan.com/schemas kompisSchema.xsd"
xmlns:ns2="http://www.ivan.com/schemas" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
    <person hasDog="true" ns2:eyeColour="brown">
        <firstName>Izzy</firstName>
        <lastName>Stradlin</lastName>
        <age>28</age>
    </person>
    <friend hasDog="false" ns2:eyeColour="blue">
        <firstName>Michael</firstName>
        <lastName>Monroe</lastName>
        <age>29</age>
    </friend>
    <friend hasDog="false" ns2:eyeColour="green">
        <firstName>Razzle</firstName>
        <lastName></lastName>
        <age>24</age>
    </friend>
</ns2:kompisRelation>
```

The formatting of the original output has been retained to show how the marshaller formats its output, when being requested to do so.

9.3 Create WSDL and Generate Service Implementation from XML Schema

Given an XML schema for a document style Web service create a WSDL file that describes the service and generate a service implementation.

Preparations

As per the instructions, we start with a given XML schema:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:tns="http://www.ivan.com/additionservicetypes"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.ivan.com/additionservicetypes" version="1.0">

    <!--
        The addValuesRequest element holds the parameters of the
        request to the web service to add two numbers.
    -->
    <xs:element name="addValuesRequest">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="value1" type="xs:double"/>
                <xs:element name="value2" type="xs:double"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!--
        The addValuesResponse element holds the result of a
        request to the web service to add two numbers.
    -->
    <xs:element name="addValuesResponse">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="sum" type="xs:double"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <!--
        The addValuesFault element holds data about error
        that occurred adding values.
    -->
    <xs:element name="addValuesFault">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="errorMsg" type="xs:string"/>
                <xs:element name="value1" type="xs:double"/>
                <xs:element name="value2" type="xs:double"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

This schema specifies the format of the in data, out data and fault data of the web service, which will contain one single operation named *addValues*.

I use Eclipse to create a Dynamic Web Project called “JAX-WS_SchemaToWS” and perform the following preparations:

- In the WEB-INF directory, create a directory named wsdl.
- In the root of the project, create a directory named build.
- In the build directory, create a directory named ant.
- In the ant directory, create a file named build.xml and paste the following into the file:

```
<?xml version="1.0"?>
<!--
    This script generates web service server and client artifacts
    from a WSDL file.
-->
<project default="main" basedir="../../">

      <!-- Default output directory for artifacts generated by wsgen. -->
      <property name="wsgen-outdir" value="${basedir}/build/wsgen-output/"/>
      <!-- Location of WSDL document used when generating artifacts. -->
      <property name="wsdl-doc"
          value="${basedir}/WebContent/WEB-INF/wsdl/AdditionService.wsdl"/>
      <!-- Directory to which generated sourcefiles will be written by wsgen. -->
      <property name="src-outdir" value="${basedir}/src/"/>
      <!-- Absolute path of the wsimport command. -->
      <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport"/>
      <!-- Location of WSDL to set in generated code. -->
      <property name="wsdl-location"
          value="http://localhost:8080/JAX-WS_SchemaToWS/calculationService?wsdl"/>

      <target name="main">
            <echo message="Generating web service server artifacts from WSDL..." />

            <!-- Create the directory in which generated artifacts will be stored. -->
            <mkdir dir="${wsgen-outdir}"/>

            <!-- Execute wsimport on the existing WSDL. -->
            <exec executable="${wsimport-cmd}">
                  <arg value="-verbose"/>

                  <!-- Specify where to write other generated files. -->
           <arg value="-d"/>
           <arg value="${wsgen-outdir}"/>

                  <!-- Specify where to write generated source files. -->
                  <arg value="-s"/>
                  <arg value="${src-outdir}"/>

                  <!--
                      Specify the WSDL location to use in the generated service
                      class.
                  -->
                  <arg value="-wsdllocation"/>
                  <arg value="${wsdl-location}"/>

                  <!-- Keep generated source files. -->
                  <arg value="-keep"/>

                  <!-- Specify WSDL to use when generating server artifacts. -->
                  <arg value="${wsdl-doc}" />
            </exec>

            <!-- Clean up generated stuff that we do not need. -->
            <delete dir="${wsgen-outdir}"/>
      </target>
</project>
```

- Review the following properties in the build.xml file and ensure that their values matches your environment: wsimport-cmd, wsdl-location
- Create a file named additionTypes.xsd in the wsdl directory and paste the given XML schema above into the file.
- Create a source package *com.ivan.client*.
- Create a source package *com.ivan.server*.

- In the WEB-INF directory, create the GlassFish specific web application deployment descriptor file named sun-web.xml, if it doesn't already exist, and paste the following into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0
Servlet 2.5//EN" "http://www.sun.com/software/appserver/dtds/sun-web-app_2_5-0.dtd">
<sun-web-app error-url="">
    <context-root>/JAX-WS_SchemaToWS</context-root>
    <class-loader delegate="true" />
    <jsp-config>
        <property name="keepgenerated" value="true">
            <description>Keep a copy of the generated servlet class' java
code.</description>
        </property>
    </jsp-config>
</sun-web-app>
```

- In the WEB-INF directory, create a file named web.xml, if it doesn't already exist, and paste the following into the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

The WSDL File

Now we are ready to write the WSDL file. It is to be named AdditionService.wsdl and is to be located in the wsdl directory in the WEB-INF directory in the project.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    WSDL file for the addition service.
-->
<wsdl:definitions
    xmlns:tns="http://www.ivan.com/AdditionService"
    xmlns:types="http://www.ivan.com/additionservicetypes"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="AdditionService"
    targetNamespace="http://www.ivan.com/AdditionService"
    xsd:schemaLocation="http://www.ivan.com/additionservicetypes additionType.xsd">

    <!--
        Here we import the schema containing the request and
        response types of the operation(s) in the web service.
    -->
    <wsdl:types>
        <xsd:schema>
            <xsd:import namespace="http://www.ivan.com/additionservicetypes"
                schemaLocation="additionType.xsd"/>
        </xsd:schema>
    </wsdl:types>

    <!--
        This message holds input data to the addValues operation.
    -->
    <wsdl:message name="additionRequest">
        <wsdl:part name="params" element="types:addValuesRequest"/>
```

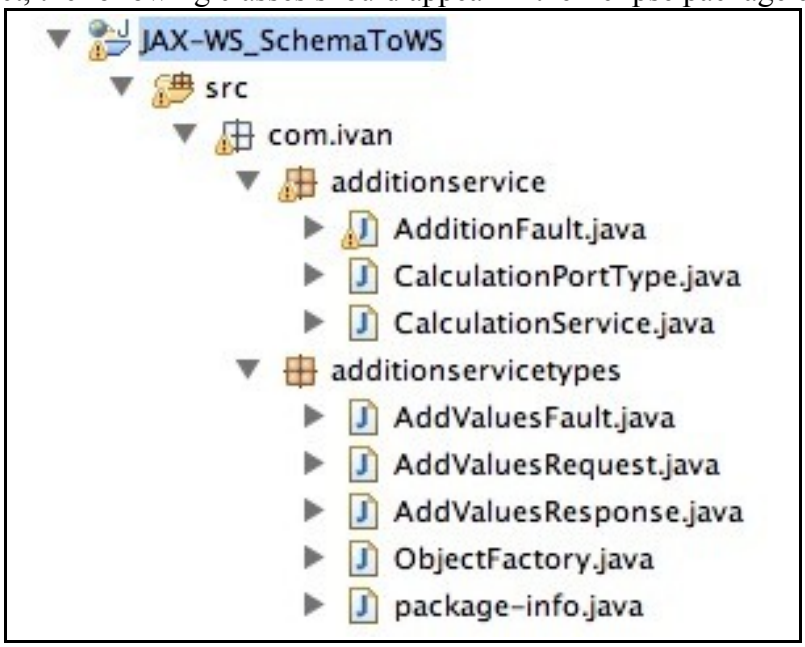
```
    </wsdl:message>

    <!--
        This message holds output data from the addValues operation.
    -->
    <wsdl:message name="additionResponse">
        <wsdl:part name="result" element="types:addValuesResponse"/>
    </wsdl:message>

    <!--
        This message holds data of a fault generated by the addValues operation.
    -->
    <wsdl:message name="additionFault">
        <wsdl:part name="error" element="types:addValuesFault"/>
    </wsdl:message>

    <!--
        This is the abstract interface of the web service.
        It only contains one single operation, addValues.
    -->
    <wsdl:portType name="calculationPortType">
        <wsdl:operation name="addValues">
            <wsdl:input message="tns:additionRequest"/>
            <wsdl:output message="tns:additionResponse"/>
            <wsdl:fault name="error" message="tns:additionFault"/>
        </wsdl:operation>
    </wsdl:portType>

    <!--
        Binding for the calculationPortType, specifying
        Document/Literal over HTTP.
    -->
    <wsdl:binding name="calculationBinding" type="tns:calculationPortType">
        <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http" style="document"/>

        <wsdl:operation name="addValues">
            <soap:operation soapAction="urn:addValues"/>
            <wsdl:input>
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"/>
            </wsdl:output>
            <wsdl:fault name="error">
                <soap:fault name="error" use="literal"/>
            </wsdl:fault>
        </wsdl:operation>

    </wsdl:binding>

    <!--
        Finally, this is the declaration of the concrete calculationService.
        If the web service is going to be deployed in GlassFish, then there
        is no need to set the location attribute of the <soap:address> element
        since GlassFish will take care of this when the service is being
        deployed.
    -->
    <wsdl:service name="calculationService">
        <wsdl:port name="calculationServicePort"
            binding="tns:calculationBinding">
            <soap:address location="TBA"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Generate Server and Client Artifacts

Now we are ready to generate server and client artifacts. These artifacts can be divided into the following categories:

- Classes representing the request, reply and fault messages.
- Exception class(es) representing fault the service may generate.
- Service interface.
  Needed both for the client and when implementing the service implementation bean.
- Service class.
  Used by the client.

These artifacts can be generated by running the build.xml Ant script created earlier. After having refreshed the project, the following classes should appear in the Eclipse package explorer:



Implement the Web Service

With the *CalculationPortType* interface generated, we can now write the server implementation bean class. Please refer to comments in the source-code for further information.

```
package com.ivan.service;

import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.xml.bind.annotation.XmlSeeAlso;

import com.ivan.additionservice.AdditionFault;
import com.ivan.additionservice.CalculationPortType;
import com.ivan.additionservicetypes.AddValuesFault;
import com.ivan.additionservicetypes.AddValuesRequest;
import com.ivan.additionservicetypes.AddValuesResponse;
import com.ivan.additionservicetypes.ObjectFactory;

/**
 * Service implementation bean for the Calculation service.
 * This looks slightly strange, because we have to implement the interface
 * generated by wsimport.
 *
 * The annotations (@WebService, @SOAPBinding and @XmlSeeAlso) were
 * copied form the generated CalculationPortType interface and the
 * following things were added to the @WebService annotation:
 * - serviceName : Specifies which <service> in the WSDL file to use.
 * - portName : Specifies which <port> in the <service> to use.
 * - wsdlLocation : Use the existing WSDL instead of generating a new one.
 *
 * @author Ivan A Krizsan
 */
@WebService(
    name = "CalculationService",
    targetNamespace = "http://www.ivan.com/AdditionService",
    serviceName = "calculationService",
    portName = "calculationServicePort",
    wsdlLocation = "WEB-INF/wsdl/AdditionService.wsdl")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
@XmlSeeAlso({ObjectFactory.class})
public class CalculationImpl implements CalculationPortType
{
    /* Constant(s): */
    private final static String ERROR_MSG = "A generated error occurred!";

    /* Instance variable(s): */
    private final ObjectFactory mObjectFactory = new ObjectFactory();

    /* (non-Javadoc)
     * @see com.ivan.additionservice.CalculationPortType#addValues(AddValuesRequest)
     */
    public AddValuesResponse addValues(final AddValuesRequest inParams)
        throws AdditionFault
    {
        /*
         * Sometimes there are errors...
         * This code is introduced to make the example more interesting,
         * since it is now possible for the operation to generate a fault.
         */
        if (Math.random() > 0.8)
        {
            AdditionFault theException;
            final AddValuesFault theFault =
                mObjectFactory.createAddValuesFault();

            theFault.setErrorMsg(ERROR_MSG);
            theFault.setValue1(inParams.getValue1());
            theFault.setValue2(inParams.getValue2());

            theException = new AdditionFault(ERROR_MSG, theFault);
            throw theException;
        }

        /*
```

```
         * Calculate the sum of the supplied values and pack the result
         * in a response object.
         */
        final double theSum = inParams.getValue1() + inParams.getValue2();
        final AddValuesResponse theResponse =
            mObjectFactory.createAddValuesResponse();
        theResponse.setSum(theSum);

        return theResponse;
    }
}
```

The web service is now ready and can be deployed to GlassFish. However, since the parameters of the *addValues* operation are wrapped in an *AddValueRequest* object, we cannot test the web service using the GlassFish built-in web service testing tool.
If you have the proper tools installed in Eclipse, you can use the Web Service Explorer to test the web service by following these steps:

- Find the URL of the deployed web service WSDL.
- Modify the WSDL file in the project to include the WSDL URL.
  Specifically, modify the <soap:address location="TBA"/> element and replace TBA with the URL.
- Right-click on the WSDL file and select Web Services → Test with Web Services Explorer.

Implement a Standalone Client

Since client artifacts were generated at the same time as the server artifacts were generated, we can implement a standalone client to the web service in the same project:

```
package com.ivan.client;

import javax.xml.ws.WebServiceException;

import com.ivan.additionservice.AdditionFault;
import com.ivan.additionservice.CalculationPortType;
import com.ivan.additionservice.CalculationService;
import com.ivan.additionservicetypes.AddValuesRequest;
import com.ivan.additionservicetypes.AddValuesResponse;
import com.ivan.additionservicetypes.ObjectFactory;

/**
 * This class implements a client that tests the CalculationService.
 *
 * @author Ivan A Krizsan
 */
public class CalculationClient
{
    /* Constant(s): */

    /* Instance variable(s): */
    private final ObjectFactory mObjectFactory = new ObjectFactory();

    private CalculationClient()
    {
        try
        {
            /*
             * Create the service and obtain a service port proxy
             * from the service.
             */
            CalculationService theService = new CalculationService();
            CalculationPortType thePort =
                theService.getCalculationServicePort();

            /*
             * Create the object holding the parameters and set parameter
             * values.
             */
            AddValuesRequest theRequest =
                mObjectFactory.createAddValuesRequest();
```

```
            theRequest.setValue1(Math.random() * 10.0);
            theRequest.setValue2(Math.random() * 10.0);

            System.out.println("Adding the values " + theRequest.getValue1()
                + " and " + theRequest.getValue2() + ":");

            /* Invoke the web service operation. */
            AddValuesResponse theResponse = thePort.addValues(theRequest);

            System.out.println("Producing the result: " + theResponse.getSum());
        } catch (final AdditionFault theException)
        {
            /*
             * Here we handle exceptions thrown by the web service.
             * Output the data from the fault information object.
             */
            System.out
                .println("*** An error occurred invoking the web service:");
            System.out.println(" Message: "
                + theException.getFaultInfo().getErrorMsg());
            System.out.println(" Parameter 1: "
                + theException.getFaultInfo().getValue1());
            System.out.println(" Parameter 2: "
                + theException.getFaultInfo().getValue2());
        } catch (final WebServiceException theException)
        {
            /*
             * This kind of problems only occur if there is something
             * seriously wrong...
             */
            System.out.println("*** A fatal error occurred:");
            System.out.println(" " + theException.getLocalizedMessage());
        }
    }

    public static void main(String[] args)
    {
        new CalculationClient();
    }
}
```

Running the client in Eclipse will produce two kinds of output; the first one indicates an addition having been successfully performed:

```
Adding the values 0.8375139684588062 and 0.0848684718637327:
Producing the result: 0.9223824403225389
```

The second output indicates an error having occurred during calculations:

```
Adding the values 4.912784555252485 and 2.7824412891137316:
*** An error occurred invoking the web service:
 Message: A generated error occurred!
 Parameter 1: 4.912784555252485
 Parameter 2: 2.7824412891137316
```

The client will produce only one kind of output per execution.

9.4 XML-Based, Document Style JAX-WS Web Service

Given a set of requirements, create code to create an XML-based, document style, Web service using the JAX-WS APIs.

This example will make use of the XJC Plugin, which is available for Eclipse and IntelliJ IDEs. This plugin aids in creating JAXB bean classes from XML schema. Please download and install this plugin, if you haven't already.

In this section we will create a XML-based, document style, calculator web service that will enable us to leverage JAX-WS web services to perform simple calculations. We will also have an opportunity to practice using JAXB to process and create XML data.
The requirements says that the service is to be XML-based and a document style service. This means that the messages sent to and from the service are complete XML fragments that can be validated against some XML schema.
In the example below, validation of incoming requests and outgoing responses will be performed.

Setting Up

As usual, I use Eclipse to develop the example and I make the following preparations:

- Create a Dynamic Web Project named JAX-WSXMLCalculator in Eclipse.
- As the Java Runtime, JRE6 is used, since the client will use JAXB classes not included in JRE5. Make sure that GlassFish is being run using JRE6, or else there will be class versioning errors!
- In the WEB-INF folder, create a folder named wsdl.
- In the wsdl folder, create a file named dummy.wsdl and insert the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
      JSR-109 Specification, section 5.3.2.2:
      "A WSDL file is required to be packaged with a Provider implementation."
      However, an empty WSDL file like this one is apparently enough.
-->
<wsdl:definitions
      targetNamespace="urn:dummy"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:bogus="urn:dummy">

</wsdl:definitions>
```

- Create the following source-code packages:
  com.ivan.beans, *com.ivan.service* and *com.ivan.client*
- Create the web deployment descriptor web.xml in the WEB-INF folder and insert the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
        id="WebApp_ID" version="2.5">

    <display-name>JAX-WSXMLCalculator</display-name>
```

```
    <listener>
        <listener-class>
            com.sun.xml.ws.transport.http.servlet.WSServletContextListener
        </listener-class>
    </listener>

    <!-- The URL pattern of the service is specified here too. See next step! -->
    <servlet>
        <display-name>XMLCalculator</display-name>
        <servlet-name>XMLCalculator</servlet-name>
        <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>XMLCalculator</servlet-name>
        <url-pattern>/xmlcalc/*</url-pattern>
    </servlet-mapping>
</web-app>
```

- Create the GlassFish-specific web service deployment descriptor sun-jaxws.xml in the WEB-INF folder and insert the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints
    xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
      version="2.0">

    <!--
        Here we specify:
        - The implementation class of the web service endpoint
        - The binding protocol of the service, HTTP in this case.
        - The URL pattern that, when appended after the server and
          context root, is used to access the service.
    -->
      <endpoint
            name="XMLCalculator"
            implementation="com.ivan.service.XMLCalculatorServiceProvider"
            binding="http://www.w3.org/2004/08/wsdl/http"
            url-pattern="/xmlcalc/*" />
</endpoints>
```

The preparations are finished, now on for the more central parts of the example.

XML Schema and JAXB Beans

First an XML schema that describes the contents of request and response messages to the XML web service is created. It is to be located in the WEB-INF/wsdl folder and named XmlServiceSchema.xsd.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    This schema specifies the format of the data received and returned
    by the XMLCalculator XML web service.
    Note that the web service does not use SOAP and the datastructures
    in this schema thus describe entire messages.
-->
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.ivan.com/xmlcalculator"
    xmlns:tns="http://www.ivan.com/xmlcalculator"
    elementFormDefault="qualified">

    <!--
        This type represents the available operations of the
        XMLCalculator web service.
    -->
    <simpleType name="operation">
        <restriction base="string">
            <enumeration value="add"/>
```

```
            <enumeration value="sub"/>
            <enumeration value="div"/>
            <enumeration value="mult"/>

            <whiteSpace value="collapse"/>
        </restriction>
    </simpleType>

    <!--
        This element describes the format of a request sent to the
        XMLCalculator web service.
    -->
    <element name="calculatorRequest">
        <complexType>
            <sequence>
                <element name="operation" type="tns:operation"/>
                <element name="value1" type="integer"/>
                <element name="value2" type="integer"/>
            </sequence>
        </complexType>
    </element>

    <!--
        This element describes the format of a response that the
        XMLCalculator web service responds with.
    -->
    <element name="calculatorResponse">
        <complexType>
            <sequence>
                <element name="result" type="long"/>
            </sequence>
        </complexType>
    </element>
</schema>
```

The XJC generator will, using the XML schema, generate the JAXB bean classes needed for this example project. I will assume that the XJC Eclipse plugin is to be used.

- Right-click the XML schema file just created and select JAXB 2.1 → Run XJC.
- In the XJC Mandatory Parameters dialog, enter the package name com.ivan.beans and select the root source directory of the project as the output directory.
- Click the Finish button.
- Refresh the project in Eclipse.

The *com.ivan.beans* package should now contain the following files:

- ObjectFactory.java
- package-info.java
- CalculatorRequest.java
- CalculatorResponse.java
- Operation.java

Service Provider Implementation

The service provider implementation is a little different, given that it has to operate on raw XML messages:

```
package com.ivan.service;

import java.math.BigInteger;
import java.net.URL;

import javax.annotation.Resource;
import javax.servlet.ServletContext;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.util.JAXBSource;
import javax.xml.transform.Source;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.ws.BindingType;
import javax.xml.ws.Provider;
import javax.xml.ws.Service;
import javax.xml.ws.ServiceMode;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.WebServiceProvider;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPBinding;

import com.ivan.beans.CalculatorRequest;
import com.ivan.beans.CalculatorResponse;
import com.ivan.beans.ObjectFactory;
import com.ivan.beans.Operation;

/**
 * Service implementation bean class of the XMLCalculator web service.
 * Since the web service is to be an XML web service, it needs to
 * operate on the raw messages received. In order to be able to accomplish
 * this, the service implementation bean implements the Provider interface
 * and specifies the service mode to MESSAGE.
 *
 * The implementation uses JAXB to unmarshal and marshal the XML data.
 * Validation is performed both when unmarshalling the request and when
 * marshalling the response.
 *
 * The service is thread-safe, meaning that multiple clients may access
 * it in parallel.
 *
 * @author Ivan A Krizsan
 */
@WebServiceProvider()
@ServiceMode(value = Service.Mode.MESSAGE)
@BindingType(value = HTTPBinding.HTTP_BINDING)
public class XMLCalculatorServiceProvider implements Provider<Source>
{
    /* Constant(s): */
    private final static String XML_SCHEMA_FILE_NAME = "XmlServiceSchema.xsd";
    private final static String XML_SCHEMA_NAMESPACE =
        "http://www.ivan.com/xmlcalculator";
    private final static String XML_SCHEMALOCATION =
        XML_SCHEMA_NAMESPACE + " " + XML_SCHEMA_FILE_NAME;
    private static final String WSDL_DIR_PATH = "/WEB-INF/wsdl/";

    /* Instance variable(s): */
    @Resource
    WebServiceContext mWSContext;
    ObjectFactory mObjectFactory;
    JAXBContext mJAXBContext;

    /**
     * Creates an instance of the service provider.
     *
     * @throws Exception If error occurs initializing.
     */
    public XMLCalculatorServiceProvider() throws Exception
```

```
    {
        mObjectFactory = new ObjectFactory();
        mJAXBContext = JAXBContext.newInstance("com.ivan.beans");
    }

    /* (non-Javadoc)
     * @see javax.xml.ws.Provider#invoke(java.lang.Object)
     */
    public Source invoke(final Source inSource)
    {
        CalculatorRequest theRequest;
        CalculatorResponse theResponse;
        BigInteger theResult;
        URL theSchemaURL;
        SchemaFactory theSchemaFactory;
        Schema theServicesSchema;
        MessageContext theMsgContext;
        ServletContext theServletContext;

        /*
         * The marshaller and unmarshaller cannot be instance variables
         * since that will cause thread-safety problems in case of
         * multiple clients accessing the service in parallel.
         */
        Unmarshaller theUnmarshaller;
        Marshaller theMarshaller;

        /* The servlet context has to be retrieved at request time. */
        theMsgContext = mWSContext.getMessageContext();
        theServletContext =
            (ServletContext)theMsgContext.get(MessageContext.SERVLET_CONTEXT);

        try
        {
            theUnmarshaller = mJAXBContext.createUnmarshaller();
            theMarshaller = mJAXBContext.createMarshaller();
            /*
             * Set a property specifying the value of the xsi:schemaLocation
             * property written to the XML document.
             */
            theMarshaller.setProperty(Marshaller.JAXB_SCHEMA_LOCATION,
                XML_SCHEMALOCATION);
            /*
             * Specifying a schema for the marshaller causes it to validate
             * the XML document produced when marshaling.
             * Note that we have to use the servlet context to retrieve
             * the URL to the XML schema.
             */
            theSchemaFactory =
                SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
            theSchemaURL =
                theServletContext.getResource(WSDL_DIR_PATH
                    + XML_SCHEMA_FILE_NAME);
            theServicesSchema = theSchemaFactory.newSchema(theSchemaURL);
            theMarshaller.setSchema(theServicesSchema);

            /*
             * Validate the XML document against its schema, as part of
             * the unmarshaling operation.
             * Having set a schema for the unmarshaler, an exception will
             * be thrown if the XML document does not pass validation.
             */
            theUnmarshaller.setSchema(theServicesSchema);

            /* Retrieve the object representing the request. */
            theRequest = (CalculatorRequest)theUnmarshaller.unmarshal(inSource);

            theResult = processRequest(theRequest);

            /* Create and marshal the result. */
            theResponse = mObjectFactory.createCalculatorResponse();
            theResponse.setResult(theResult.longValue());

            return new JAXBSource(theMarshaller, theResponse);
        } catch (Exception theException)
        {
            theException.printStackTrace();
```

```
        }

        return null;
    }

    /**
     * Handles a request to the calculator by inspecting the operation
     * and delegating to appropriate handler.
     *
     * @param inRequest Object holding request parameters.
     * @return Result of calculation.
     */
    private BigInteger processRequest(final CalculatorRequest inRequest)
    {
        BigInteger theResult = null;

        if (inRequest.getOperation() == Operation.ADD)
        {
            theResult =
                doAddition(inRequest.getValue1(), inRequest.getValue2());
        } else if (inRequest.getOperation() == Operation.SUB)
        {
            theResult =
                doSubtraction(inRequest.getValue1(), inRequest.getValue2());
        } else if (inRequest.getOperation() == Operation.DIV)
        {
            theResult =
                doDivision(inRequest.getValue1(), inRequest.getValue2());
        } else if (inRequest.getOperation() == Operation.MULT)
        {
            theResult =
                doMultiplication(inRequest.getValue1(), inRequest.getValue2());
        } else
        {
            throw new Error("Illegal operation in calculator request!");
        }

        return theResult;
    }

    private BigInteger doAddition(final BigInteger inValue1,
        final BigInteger inValue2)
    {
        return inValue1.add(inValue2);
    }

    private BigInteger doSubtraction(final BigInteger inValue1,
        final BigInteger inValue2)
    {
        return inValue1.subtract(inValue2);
    }

    private BigInteger doDivision(final BigInteger inValue1,
        final BigInteger inValue2)
    {
        return inValue1.divide(inValue2);
    }

    private BigInteger doMultiplication(final BigInteger inValue1,
        final BigInteger inValue2)
    {
        return inValue1.multiply(inValue2);
    }
}
```

Now the web service can be deployed to GlassFish.

XML Web Service Client

Outside the requirements, we will create a client interacting with the XML-based calculator web service created above. The client is created in the web service server project and is only run from within Eclipse.
Please note that the client will request user input in the console window!

```
package com.ivan.client;

import java.io.BufferedReader;
import java.io.File;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.net.MalformedURLException;
import java.util.Map;

import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.util.JAXBSource;
import javax.xml.namespace.QName;
import javax.xml.transform.Source;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.http.HTTPBinding;

import com.ivan.beans.CalculatorRequest;
import com.ivan.beans.CalculatorResponse;
import com.ivan.beans.ObjectFactory;
import com.ivan.beans.Operation;

/**
 * This class implements a client to the XMLCalculator web service.
 * Accepting user input regarding what operation to perform and on
 * which numbers to perform it, then invoking the web service to
 * obtain a result which is printed to the console.
 *
 * @author Ivan A Krizsan
 */
public class XMLCalculatorClient
{
    /* Constant(s): */
    private final static String SERVICE_URL =
        "http://localhost:8080/JAX-WSXMLCalculator/xmlcalc/";
    private final static String XML_SCHEMA_FILE_NAME =
        "WebContent/WEB-INF/wsdl/XmlServiceSchema.xsd";
    private final static String XML_SCHEMA_NAMESPACE =
        "http://www.ivan.com/xmlcalculator";
    private final static String XML_SCHEMALOCATION =
        XML_SCHEMA_NAMESPACE + " " + XML_SCHEMA_FILE_NAME;

    /* Instance variable(s): */
    private ObjectFactory mObjectFactory;
    private JAXBContext mJAXBContext;
    private Unmarshaller mUnmarshaller;
    private Marshaller mMarshaller;

    public static void main(String[] args)
    {
        try
        {
            new XMLCalculatorClient();
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private XMLCalculatorClient() throws Exception
    {
```

```
        /* Ask the user what to do and what numbers to use. */
        BufferedReader theUserInput =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.println("What operation do you want to perform " +
            "(Addition = " + Operation.ADD.value() + ", " +
            "Subtraction = " + Operation.SUB.value() + ", " +
            "Division = " + Operation.DIV.value() + ", " +
            "Multiplication = " + Operation.MULT.value() + ")?");
        String theOpString = theUserInput.readLine();
        Operation theOp = Operation.fromValue(theOpString);

        System.out.println("Give the first integer number: ");
        String theValString1 = theUserInput.readLine();

        System.out.println("Give the second integer number: ");
        String theValString2 = theUserInput.readLine();

        init();

        Dispatch<Source> theCalcDispatcher = createServiceDispatcher();

        /* Prepare the request object. */
        CalculatorRequest theRequest = mObjectFactory.createCalculatorRequest();
        theRequest.setOperation(theOp);
        theRequest.setValue1(new BigInteger(theValString1));
        theRequest.setValue2(new BigInteger(theValString2));

        JAXBSource theReqSource = new JAXBSource(mMarshaller, theRequest);

        /* Invoke the XMLCalculator web service. */
        Source theResultSource = theCalcDispatcher.invoke(theReqSource);

        /* Unmarshal and print the result. */
        CalculatorResponse theResult =
            (CalculatorResponse)mUnmarshaller.unmarshal(theResultSource);

        String theVerb = operationToVerb(theRequest.getOperation());
        System.out.println(theVerb + " the numbers " + theRequest.getValue1()
            + " and " + theRequest.getValue2() + " produces the " + "result "
            + theResult.getResult());
    }

    private String operationToVerb(final Operation inOp)
    {
        String theVerb = "[unknown verb]";

        if (inOp == Operation.ADD)
        {
            theVerb = "Adding";
        } else if (inOp == Operation.SUB)
        {
            theVerb = "Subtracting";
        } else if (inOp == Operation.DIV)
        {
            theVerb = "Dividing";
        } else if (inOp == Operation.MULT)
        {
            theVerb = "Multiplying";
        }

        return theVerb;
    }

    private Dispatch<Source> createServiceDispatcher()
        throws MalformedURLException
    {
        QName theDummyQName = new QName("", "");

        /*
         * Create the service and obtain a dispatcher used to invoke
         * the XML Calculator.
         */
        Service theXmlCalcService = Service.create(theDummyQName);
        theXmlCalcService.addPort(theDummyQName, HTTPBinding.HTTP_BINDING,
            SERVICE_URL);
        Dispatch<Source> theCalcDispatcher =
            theXmlCalcService.createDispatch(theDummyQName, Source.class,
```

```
                Service.Mode.MESSAGE);

        /*
         * Set the HTTP request method in the request context of the
         * dispatcher.
         */
        Map<String, Object> theRequestContext =
            theCalcDispatcher.getRequestContext();
        theRequestContext.put(MessageContext.HTTP_REQUEST_METHOD, "POST");

        return theCalcDispatcher;
    }

    private void init() throws Exception
    {
        mObjectFactory = new ObjectFactory();
        mJAXBContext = JAXBContext.newInstance("com.ivan.beans");
        mUnmarshaller = mJAXBContext.createUnmarshaller();
        mMarshaller = mJAXBContext.createMarshaller();
        /*
         * Set a property specifying the value of the
         * xsi:schemaLocation property written to the XML document.
         */
        mMarshaller.setProperty(Marshaller.JAXB_SCHEMA_LOCATION,
            XML_SCHEMALOCATION);
        /*
         * Specifying a schema for the marshaller causes it to
         * validate the XML document produced when marshaling.
         */
        SchemaFactory theSchemaFactory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema theServicesSchema =
            theSchemaFactory.newSchema(new File(XML_SCHEMA_FILE_NAME));
        mMarshaller.setSchema(theServicesSchema);
    }
}
```

When run, the client will produce console output like in this example:

```
What operation do you want to perform (Addition = add, Subtraction = sub, Division =
div, Multiplication = mult)?
mult
Give the first integer number:
5
Give the second integer number:
8
Multiplying the numbers 5 and 8 produces the result 40
```

If we look at the data exchanged between the client and the server, we can see that a request has the following format (text has been formatted):

```
<calculatorRequest
    xmlns="http://www.ivan.com/xmlcalculator"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.ivan.com/xmlcalculator
        WebContent/WEBINF/wsdl/XmlServiceSchema.xsd">
    <operation>mult</operation>
    <value1>5</value1>
    <value2>8</value2>
</calculatorRequest>
```

The response to the above request has the following format (text has been formatted):

```
<calculatorResponse
    xmlns="http://www.ivan.com/xmlcalculator"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/xmlcalculator XmlServiceSchema.xsd">
    <result>40</result>
</calculatorResponse>
```

9.5 SOAP Logging

Implement a SOAP logging mechanism for testing and debugging a Web service application using Java EE Web Service APIs.

References:
Web Services for JavaEE v1.2 (JSR-109), chapter 6.
JAX-WS 2.1 Specification, chapter 9.

In order to be able to intercept messages sent to, and received by, a web service application, we can use handlers. A handler works like a filter, allowing access and modification of messages and corresponding message contexts that passes through the handler. The JAX-WS API supports two types of handlers:

- Logical Handlers
  Independent of protocol binding, treats the payload of a message as arbitrary XML and does not provide access to protocol-specific parts of messages.
  See *javax.xml.ws.handler.LogicalHandler<C extends LogicalMessageContext>* in the JAX-WS API.
- Protocol Handlers
  Considers protocol binding, treats the payload of a message as, for instance, a SOAP message. In the case of SOAP, the handler receives a *SOAPMessageContext* and can access the header blocks of the SOAP message.
  See *javax.xml.ws.handler.soap.SOAPHandler<T extends SOAPMessageContext>* in the JAX-WS API.

Since a SOAP logging mechanism is to be implemented, a protocol handler for SOAP will be implemented in this example. A logging handler will be added both on the server and on the client side, showing how to configure handers using the @HandlerChain annotation (server side) and how to configure them at runtime using the *HandlerResolver* interface (client side).

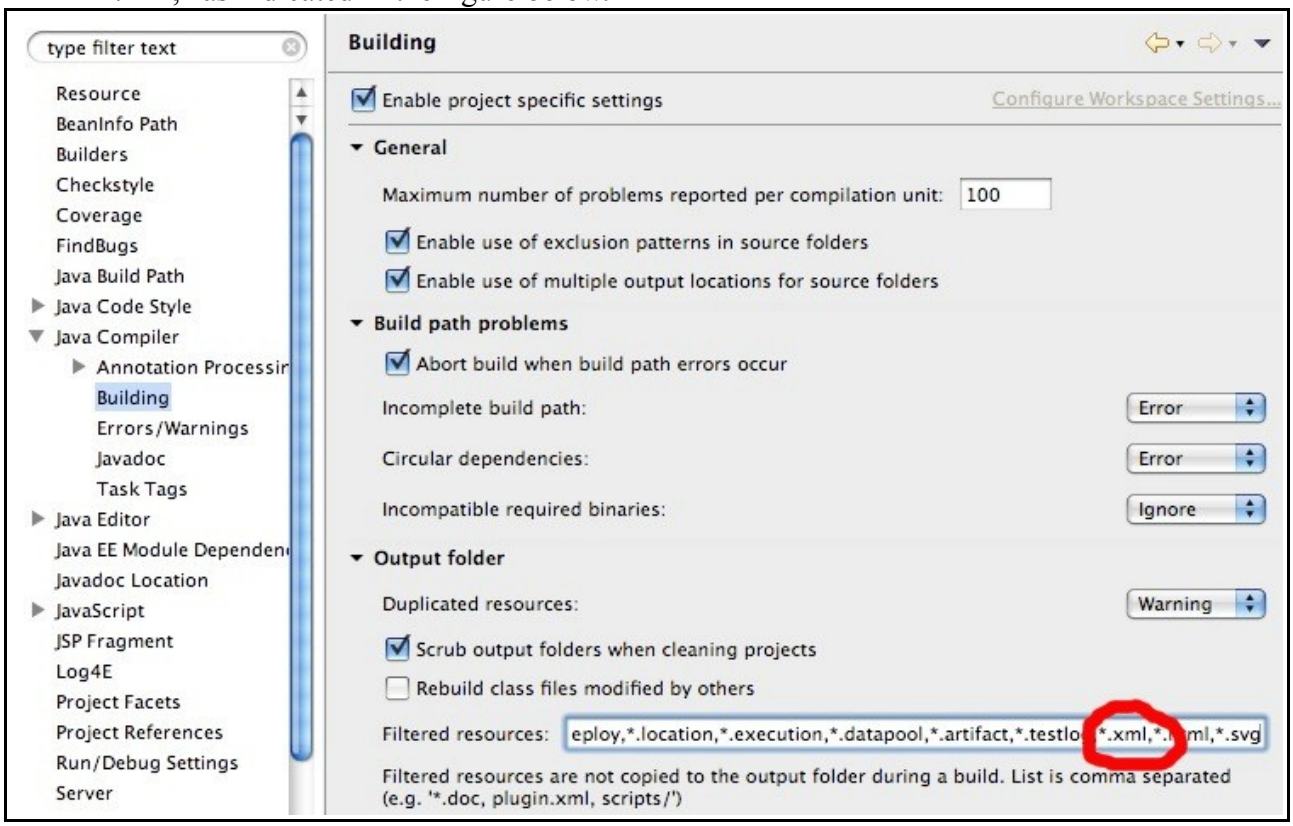
Preparations

The web service used in this example is a simple Calculator web service that adds two numbers.

- Create a dynamic web project with the name "JAX-WSCalculatorSOAPLogging" in Eclipse.
- Make sure that you have a web.xml deployment descriptor.
  We won't use it, but GlassFish may have problems recognizing the project type, which is basically a servlet project, if it is not present. No special configuration is required.
- Create the source code package com.ivan.service.
- In the above package, create the *Calculator* class:

```
package com.ivan.service;

import javax.jws.HandlerChain;
import javax.jws.WebService;

@WebService()
public class Calculator
{
    public int divide(final int inNumber1, final int inNumber2)
    {
        System.out.println("Dividing " + inNumber1 + " and " + inNumber2);
        return inNumber1 / inNumber2;
    }
}
```

- Set the resource filter of the Eclipse project NOT to filter out XML files by removing the ".xml," as indicated in the figure below.



The Calculator web service can now be deployed and tested using the web service testing tool in GlassFish.

Server Side Logging

To add server side logging of SOAP messages received and sent by the Calculator web service, we modify the example project as follows:

- Annotate the *Calculator* class with the @HandlerChain annotation.
  In the annotation, using the file attribute, we have to specify the location of the XML file that contains the handler chain configuration.
  The location can either be an URL or a relative path from the location of the class file of the class containing the annotation.
  Example: @HandlerChain(file="handlerchain.xml")
- Create a file handlerchain.xml containing the handler chain configuration in the com.ivan.service source package.
  In this example, the file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains xmlns="http://java.sun.com/xml/ns/javaee">
    <handler-chain>
        <handler>
            <handler-name>SOAPLogger</handler-name>
            <handler-class>com.ivan.handler.SOAPLogger</handler-class>
        </handler>
    </handler-chain>
</handler-chains>
```

- Create a source package com.ivan.handler
- Create the class implementing the logging handler:

```
package com.ivan.handlers;

import java.util.Collections;
import java.util.Map;
import java.util.Set;

import javax.servlet.http.HttpServletRequest;
import javax.xml.namespace.QName;
import javax.xml.soap.SOAPMessage;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.handler.soap.SOAPHandler;
import javax.xml.ws.handler.soap.SOAPMessageContext;

/**
 * This class implements a handler that logs SOAP messages passing
 * through it. The messages are not altered in any way. Data is output
 * to the console.
 *
 * @author Ivan A Krizsan
 */
public class SOAPLogHandler implements SOAPHandler<SOAPMessageContext>
{

    /*
     * (non-Javadoc)
     *
     * @see javax.xml.ws.handler.soap.SOAPHandler#getHeaders()
     */
    public Set<QName> getHeaders()
    {
        /*
         * Returning an empty set means the handler does not process
         * any headers.
         */
        return Collections.EMPTY_SET;
    }

    /*
     * (non-Javadoc)
     *
     * @seejavax.xml.ws.handler.Handler#close(javax.xml.ws.handler.
     * MessageContext)
```

```
     */
    public void close(final MessageContext inMessageContext)
    {
        /* Does nothing. */
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * javax.xml.ws.handler.Handler#handleFault(javax.xml.ws.handler
     * .MessageContext)
     */
    public boolean handleFault(final SOAPMessageContext inSOAPMsgContext)
    {
        System.out.println("********** A SOAP fault was received:");
        logSOAPMessage(inSOAPMsgContext);

        /*
         * Returning true means that message processing is to
         * continue.
         */
        return true;
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * javax.xml.ws.handler.Handler#handleMessage(javax.xml.ws.handler
     * .MessageContext)
     */
    public boolean handleMessage(final SOAPMessageContext inSOAPMsgContext)
    {
        System.out.println("********** A SOAP message was received:");
        logSOAPMessage(inSOAPMsgContext);

        /*
         * Returning true means that message processing is to
         * continue.
         */
        return true;
    }

    private void logSOAPMessage(final SOAPMessageContext inSOAPMsgContext)
    {
        Boolean theOutboundFlag;
        Map theHTTPRequestHdrs;
        String theDirectionString;
        String theHTTPMethod;
        Integer theResponseCode;
        Map theHTTPResponseHdrs;
        String theReqPathInfo;
        String theQueryString;
        QName theWSDLService;
        QName theWSDLInterface;
        QName theWSDLOperation;
        QName theWSDLPort;
        SOAPMessage theMsg;
        HttpServletRequest theServletReq;

        /* Retrive data to log from the message context. */
        theServletReq =
            (HttpServletRequest)inSOAPMsgContext
                .get(MessageContext.SERVLET_REQUEST);
        theOutboundFlag =
            (Boolean)inSOAPMsgContext
                .get(MessageContext.MESSAGE_OUTBOUND_PROPERTY);
        theHTTPRequestHdrs =
            (Map)inSOAPMsgContext.get(MessageContext.HTTP_REQUEST_HEADERS);
        theHTTPMethod =
            (String)inSOAPMsgContext.get(MessageContext.HTTP_REQUEST_METHOD);
        theResponseCode =
            (Integer)inSOAPMsgContext.get(MessageContext.HTTP_RESPONSE_CODE);
        theHTTPResponseHdrs =
            (Map)inSOAPMsgContext.get(MessageContext.HTTP_RESPONSE_HEADERS);
        theReqPathInfo = (String)inSOAPMsgContext.get(MessageContext.PATH_INFO);
```

```
        theQueryString =
            (String)inSOAPMsgContext.get(MessageContext.QUERY_STRING);
        theWSDLService =
            (QName)inSOAPMsgContext.get(MessageContext.WSDL_SERVICE);
        theWSDLInterface =
            (QName)inSOAPMsgContext.get(MessageContext.WSDL_INTERFACE);
        theWSDLOperation =
            (QName)inSOAPMsgContext.get(MessageContext.WSDL_OPERATION);
        theWSDLPort = (QName)inSOAPMsgContext.get(MessageContext.WSDL_PORT);
        theMsg = inSOAPMsgContext.getMessage();

        /* Print logging data to console. */
        if (theServletReq != null)
        {
            System.out.println("     Request URL from servlet request: "
                + theServletReq.getRequestURL());
        }
        theDirectionString = (theOutboundFlag) ? "Outbound" : "Inbound";
        System.out.println("     Message direction: " + theDirectionString);
        System.out.println("     HTTP request headers: " + theHTTPRequestHdrs);
        System.out.println("     HTTP request method: " + theHTTPMethod);
        System.out
            .println("     HTTP response headers: " + theHTTPResponseHdrs);
        System.out.println("     HTTP response code: " + theResponseCode);
        System.out.println("     HTTP request path info: " + theReqPathInfo);
        System.out.println("     HTTP query string: " + theQueryString);
        System.out.println("     WSDL service: " + theWSDLService);
        System.out.println("     WSDL interface: " + theWSDLInterface);
        System.out.println("     WSDL operation: " + theWSDLOperation);
        System.out.println("     WSDL port: " + theWSDLPort);

        try
        {
            theMsg.writeTo(System.out);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }
}
```

The project can now be deployed and tested using the GlassFish web service test tool. Note that if we enter legal data, such as 144 and 6, we will see the following output on the console from our SOAP logger:

```
INFO: ********** A SOAP message was received:
INFO:      Request URL from servlet request: http://localhost:8080/JAX-
WSSOAPLogging/CalculatorService
INFO:      Message direction: Inbound
INFO:      HTTP request headers: {Host=[localhost:8080], Content-length=[208], Content-
type=[text/xml;charset="utf-8"], Accept=[text/xml, multipart/related, text/html,
image/gif, image/jpeg, *; q=.2, */*; q=.2], Connection=[keep-alive], Soapaction=[""],
User-agent=[JAX-WS RI 2.1.2_01-hudson-189-]}
INFO:      HTTP request method: POST
INFO:      HTTP response headers: null
INFO:      HTTP response code: 0
INFO:      HTTP request path info: null
INFO:      HTTP query string: null
INFO:      WSDL service: {http://service.ivan.com/}CalculatorService
INFO:      WSDL interface: {http://service.ivan.com/}Calculator
INFO:      WSDL operation: null
INFO:      WSDL port: {http://service.ivan.com/}CalculatorPort
INFO: <S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:divide
xmlns:ns2="http://service.ivan.com/"><arg0>144</arg0><arg1>6</arg1></ns2:divide></S:Body
></S:Envelope>
INFO: Dividing 144 and 6
INFO: ********** A SOAP message was received:
INFO:      Request URL from servlet request: http://localhost:8080/JAX-
WSSOAPLogging/CalculatorService
INFO:      Message direction: Outbound
INFO:      HTTP request headers: {Host=[localhost:8080], Content-length=[208], Content-
type=[text/xml;charset="utf-8"], Accept=[text/xml, multipart/related, text/html,
image/gif, image/jpeg, *; q=.2, */*; q=.2], Connection=[keep-alive], Soapaction=[""],
User-agent=[JAX-WS RI 2.1.2_01-hudson-189-]}
```

```
INFO:      HTTP request method: POST
INFO:      HTTP response headers: null
INFO:      HTTP response code: 0
INFO:      HTTP request path info: null
INFO:      HTTP query string: null
INFO:      WSDL service: {http://service.ivan.com/}CalculatorService
INFO:      WSDL interface: {http://service.ivan.com/}Calculator
INFO:      WSDL operation: null
INFO:      WSDL port: {http://service.ivan.com/}CalculatorPort
INFO: <S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:divideResponse
xmlns:ns2="http://service.ivan.com/"><return>24</return></ns2:divideResponse></S:Body></
S:Envelope>
```

We see that there is an incoming and an outgoing message.
If we however change the parameters to the web service, supplying zero as the second parameter, we can see that there is an incoming message and a fault message (the stacktrace and the fault message has been shortened to preserve space):

```
INFO: ********** A SOAP message was received:
INFO:      Request URL from servlet request: http://localhost:8080/JAX-
WSSOAPLogging/CalculatorService
INFO:      Message direction: Inbound
INFO:      HTTP request headers: {Host=[localhost:8080], Content-length=[206], Content-
type=[text/xml;charset="utf-8"], Accept=[text/xml, multipart/related, text/html,
image/gif, image/jpeg, *; q=.2, */*; q=.2], Connection=[keep-alive], Soapaction=[""],
User-agent=[JAX-WS RI 2.1.2_01-hudson-189-]}
INFO:      HTTP request method: POST
INFO:      HTTP response headers: null
INFO:      HTTP response code: 0
INFO:      HTTP request path info: null
INFO:      HTTP query string: null
INFO:      WSDL service: {http://service.ivan.com/}CalculatorService
INFO:      WSDL interface: {http://service.ivan.com/}Calculator
INFO:      WSDL operation: null
INFO:      WSDL port: {http://service.ivan.com/}CalculatorPort
INFO: <S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:divide
xmlns:ns2="http://service.ivan.com/"><arg0>6</arg0><arg1>0</arg1></ns2:divide></S:Body><
/S:Envelope>
INFO: Dividing 6 and 0
SEVERE: / by zero
java.lang.ArithmeticException: / by zero
      at com.ivan.service.Calculator.divide(Calculator.java:30)
      ...

INFO: ********** A SOAP fault was received:
INFO:      Request URL from servlet request: http://localhost:8080/JAX-
WSSOAPLogging/CalculatorService
INFO:      Message direction: Outbound
INFO:      HTTP request headers: {Host=[localhost:8080], Content-length=[206], Content-
type=[text/xml;charset="utf-8"], Accept=[text/xml, multipart/related, text/html,
image/gif, image/jpeg, *; q=.2, */*; q=.2], Connection=[keep-alive], Soapaction=[""],
User-agent=[JAX-WS RI 2.1.2_01-hudson-189-]}
INFO:      HTTP request method: POST
INFO:      HTTP response headers: null
INFO:      HTTP response code: 0
INFO:      HTTP request path info: null
INFO:      HTTP query string: null
INFO:      WSDL service: {http://service.ivan.com/}CalculatorService
INFO:      WSDL interface: {http://service.ivan.com/}Calculator
INFO:      WSDL operation: null
INFO:      WSDL port: {http://service.ivan.com/}CalculatorPort
INFO: <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Body><ns2:Fault
xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns3="http://www.w3.org/2003/05/soap-
envelope"><faultcode>ns2:Server</faultcode><faultstring>/ by
zero</faultstring><detail><ns2:exception xmlns:ns2="http://jax-ws.dev.java.net/"
class="java.lang.ArithmeticException" note="To disable this feature, set
com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system property to
false"><message>/ by zero</message><ns2:stackTrace><ns2:frame
class="com.ivan.service.Calculator" file="Calculator.java" line="30" method="divide"/>
...
</ns2:stackTrace></ns2:exception></detail></ns2:Fault></S:Body></S:Envelope>
```

Client Side Logging

To show how to log SOAP messages on the client side, a client to the Calculator web service will be created in the same project.

Service Modifications

To be able to create a client, we first need to modify the web service slightly:

- In the com.ivan.service package, create an interface as follows:

```
package com.ivan.service;

import javax.jws.WebService;

@WebService
public interface CalculatorInterface
{
    int divide(final int inNumber1, final int inNumber2);

}
```

- Modify the @WebService annotation in the *Calculator* class to read: @WebService(endpointInterface="com.ivan.service.CalculatorInterface")
- Redeploy the Calculator web service.

Basic Client Implementation

The client does not need any generated artifacts. It only depends on the *CalculatorInterface* created above. We'll also have an opportunity to practice using DOM to extract the result from the response message.

```
package com.ivan.client;

import java.io.ByteArrayOutputStream;
import java.io.StringReader;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.ws.Dispatch;
import javax.xml.ws.Service;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

/**
 * Client to the Calculator web service.
 * Note that the payload of the request is hardcoded in the client
 * and needs to be updated if there are changes to the service!
 *
 * @author Ivan A Krizsan
 */
public class CalculatorClient
{
    /* Constant(s): */
    private final static String SERVICE_URL =
        "http://localhost:8080/JAX-WSSOAPLogging/CalculatorService";
```

```
    private final static String WSDL_URL = SERVICE_URL + "?wsdl";
    private final static String WSDL_NAMESPACE = "http://service.ivan.com/";
    private final static String SERVICE_NAME = "CalculatorService";
    private final static String PORT_NAME = "CalculatorPort";
    private final static int FIRST_NUM = 166;
    private final static int SECOND_NUM = 4;
    private final static String DIVIDE_PAYLOAD =
        "<ns2:divide
xmlns:ns2=\"http://service.ivan.com/\"><arg0>p1</arg0><arg1>p2</arg1></ns2:divide>";
    private final static String RETURN_ELEMENT_NAME = "return";


    public static void main(String[] args)
    {
        try
        {
            Service theCalculatorService;
            DocumentBuilder theDocBuilder =
                DocumentBuilderFactory.newInstance().newDocumentBuilder();

            /* Create service object for the Calculator service. */
            URL theWSDLURL = new URL(WSDL_URL);
            QName theServiceName = new QName(WSDL_NAMESPACE, SERVICE_NAME);
            QName thePortName = new QName(WSDL_NAMESPACE, PORT_NAME);
            theCalculatorService = Service.create(theWSDLURL, theServiceName);

            /*
             * Create a dispatch object for the Calculator service.
             * Note that payload mode is used, in order to only have to
             * prepare the message that goes in the SOAP body and
             * not to have to care about the SOAP envelope itself.
             */
            Dispatch<Source> theDispatcher =
                theCalculatorService.createDispatch(thePortName, Source.class,
                    Service.Mode.PAYLOAD);

            /*
             * Prepare the message payload to be sent by setting parameters
             * in the pre-prepared payload XML fragment.
             * Must also create a Source, taking its data from the string
             * containing the XML fragment, that is supplied when invoking
             * the service.
             */
            String thePayload = DIVIDE_PAYLOAD.replaceAll("p1", "" + FIRST_NUM);
            thePayload = thePayload.replaceAll("p2", "" + SECOND_NUM);
            StringReader thePayloadReader = new StringReader(thePayload);
            Source thePayloadSource = new StreamSource(thePayloadReader);

            /* Invoke the Calculator service. */
            Source theResultSource = theDispatcher.invoke(thePayloadSource);

            /*
             * Extract the result.
             * First transform the received result to a string that
             * then is parsed, creating a DOM document.
             * The DOM document is then searched for the element containing
             * the division result and the result string is retrieved.
             */
            String theResultString = sourceToString(theResultSource);
            Document theDOMDoc =
                theDocBuilder.parse(new InputSource(new StringReader(
                    theResultString)));
            NodeList theNodeList =
                theDOMDoc.getElementsByTagName(RETURN_ELEMENT_NAME);
            Node theReturnNode = theNodeList.item(0);
            String theResult = theReturnNode.getFirstChild().getNodeValue();

            /* Output the result to the console. */
            System.out.println(FIRST_NUM + " / " + SECOND_NUM + " = "
                + theResult);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    private static String sourceToString(final Source inSource)
```

```
        throws Exception
    {
        ByteArrayOutputStream theOutStream = new ByteArrayOutputStream();
        Transformer theTransformer =
            TransformerFactory.newInstance().newTransformer();
        Result theTransformerResult = new StreamResult(theOutStream);
        theTransformer.transform(inSource, theTransformerResult);

        theOutStream.flush();
        return new String(theOutStream.toByteArray());
    }
}
```

The client can now be executed and will output the following result to the console:

```
166 / 4 = 41
```

Note that the Calculator service performs integer divisions, so the above result is correct.

Adding Logging

Handlers on the client side can be configured either using the @HandlerChain annotation or programmatically. Since the client developed above is a standalone client, the latter approach will be used in this example.

- Implement a handler resolver.
  A handler resolver will, at the time when a proxy or dispatch object is created, be queried for the handler chain to be used for the proxy or dispatch object.
  Note that we can reuse the *SOAPLogHandler* class used on the server side.

```
package com.ivan.handlers;

import java.util.ArrayList;
import java.util.List;

import javax.xml.ws.handler.Handler;
import javax.xml.ws.handler.HandlerResolver;
import javax.xml.ws.handler.PortInfo;


/**
 * This class implements a handler resolver used when programmatically
 * setting handlers in a web service client.
 *
 * @author Ivan A Krizsan
 */
public class ClientHandlerResolver implements HandlerResolver
{
    /* (non-Javadoc)
     * @see
javax.xml.ws.handler.HandlerResolver#getHandlerChain(javax.xml.ws.handler.PortInfo)
     */
    public List<Handler> getHandlerChain(final PortInfo inPortInfo)
    {
        /*
         * Ignore the port info telling the handler resolver for which
         * port a handler chain is to be retrieved, since we only
         * have one single port in this example.
         */
        ArrayList<Handler> theHandlerList = new ArrayList<Handler>();
        theHandlerList.add(new SOAPLogHandler());
        return theHandlerList;
    }
}
```

- Modify the *CalculatorClient* class, adding the last two lines in the following code snippet:

```
    /* Create service object for the Calculator service. */
    URL theWSDLURL = new URL(WSDL_URL);
    QName theServiceName = new QName(WSDL_NAMESPACE, SERVICE_NAME);
    QName thePortName = new QName(WSDL_NAMESPACE, PORT_NAME);
    theCalculatorService = Service.create(theWSDLURL, theServiceName);

    /* Set the handler resolver of the service. */
    theCalculatorService.setHandlerResolver(new ClientHandlerResolver());
```

The client can now be run and will produce the following console output:

```
********** A SOAP message was received:
     Message direction: Outbound
     HTTP request headers: null
     HTTP request method: null
     HTTP response headers: null
     HTTP response code: null
     HTTP request path info: null
     HTTP query string: null
     WSDL service: {http://service.ivan.com/}CalculatorService
     WSDL interface: {http://service.ivan.com/}Calculator
     WSDL operation: null
     WSDL port: {http://service.ivan.com/}CalculatorPort
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:divide
xmlns:ns2="http://service.ivan.com/"><arg0>166</arg0><arg1>4</arg
1></ns2:divide></S:Body></S:Envelope>********** A SOAP message was received:
     Message direction: Inbound
     HTTP request headers: null
     HTTP request method: null
     HTTP response headers: {Date=[Wed, 22 Apr 2009 21:10:35 GMT], X-powered-
by=[Servlet/2.5], Content-type=[text/xml;charset="utf-8"], Transfer-encoding=[chunk
ed], Server=[Sun Java System Application Server 9.1_01], null=[HTTP/1.1 200 OK]}
     HTTP response code: 200
     HTTP request path info: null
     HTTP query string: null
     WSDL service: {http://service.ivan.com/}CalculatorService
     WSDL interface: {http://service.ivan.com/}Calculator
     WSDL operation: null
     WSDL port: {http://service.ivan.com/}CalculatorPort
<S:Envelope
xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header/><S:Body><ns2:divideRespon
se xmlns:ns2="http://service.ivan.com/"><return>41</return><
/ns2:divideResponse></S:Body></S:Envelope>166 / 4 = 41
```

9.6 Web Service Client Error Handling

Given a set of requirements, create code to handle system and service exceptions and faults received by a Web services client.

Exception Mapping

References:
JAX-WS 2.1 Specification, sections 3.7, 4.2, 6.4, 10.2.2

Service Exceptions and System Exceptions

Service specific exceptions are all checked exceptions, except *java.rmi.RemoteException* and all its subclasses. Service specific exceptions should, in the WSDL file for the service, have corresponding fault declared in the corresponding operation, both in the abstract interface and in the binding of the service.
Exceptions that are not service exceptions must not be treated as service exceptions and must not be mapped to WSDL faults.

Web Service Client Exception View

A client of a JAX-WS web service invokes methods on a Service Endpoint Interface, which may throw *javax.xml.ws.WebServiceException* and any number of service exceptions.

Type of Error	Server Side Exception	Client Exception
Error occurring during remote invocation of service implementation.	Service specific exception.	Service exception.
Error occuring during remote invocation of service implementation.	Non service specific exception, that is, unchecked exception or *RemoteException*.	*ProtocolException* or subclass, depending on protocol used.
Exception thrown during handler processing on client side. Exception is a *WebServiceException* or subclass.	N/A	Exception is re-thrown as is.
Exception thrown during handler processing on client side. Exception is not a *WebServiceException* or subclass thereof.	N/A	Exception is wrapped in a *WebServiceException* with the original exception as cause and re-thrown.
Other client side exceptions.	N/A	Exception is wrapped in a *WebServiceException* with the original exception as cause and re-thrown.

Error Service

We'll now implement a service that will produce different kinds of errors, depending on a selector parameter, and, later, a client invoking the error service.
As usual, the service will be developed in a dynamic web project in Eclipse and deployed to GlassFish. The name of my service project is JAX-WSExceptionFaultHandling.

Error Service Implementation

The part of the service we have to write only consists of a single class, which is implemented as follows:

```
package com.ivan.service;

import java.util.Date;

import javax.jws.WebService;

/**
 * This class implements a web service endpoint that, given certain
 * input, generates service exceptions, system exceptions or a message.
 *
 * @author Ivan A Krizsan
 */
@WebService()
public class ErrorService
{
    private final static int SYS_EX_SELECTOR = 1;
    private final static int SERVICE_EX_SELECTOR = 2;

    public String generate(final int inSelector) throws Exception
    {
        Date theNow = new Date();

        switch (inSelector)
        {
            case SYS_EX_SELECTOR:
                throw new IllegalArgumentException("A system exception at "
                    + theNow);

            case SERVICE_EX_SELECTOR:
                throw new Exception("A service exception at " + theNow);
        }

        return "The current time is: " + theNow;
    }
}
```

The service may now be deployed and tested.

First Error Client

The first client of the error service consists of generated classes and interfaces, a main class and an Ant script used to create the generated artifacts. The client was developed in a separate, plain Java, project.

Ant Script

The script is located in the root of the project, in a file named build.xml. Additionally, a directory with the name wsimport_generated must also be created in the project's root.
Note that you may have to modify the location of the wsimport command according to your system configuration.

```
<?xml version="1.0"?>

<project default="main" basedir=".">

      <property name="class-dir" value="${basedir}/build/classes" />
      <property name="wsimport-outdir" value="${basedir}/wsimport_generated" />
      <property name="gen-classdir" value="${wsimport-outdir}/com/" />
      <property name="src-outdir" value="${basedir}/src/" />
      <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport" />
      <property name="wsdl-location" value="http://localhost:8080/JAX-
WSExceptionFaultHandling/ErrorServiceService?wsdl" />

      <echo message="calling the client generation task wsimport" />
      <target name="main">
            <exec executable="${wsimport-cmd}">
                  <arg value="-verbose" />

                  <!-- Specify where to write other generated files. -->
                  <arg value="-d" />
                  <arg value="${wsimport-outdir}" />

                  <!-- Specify where to write generated source files. -->
                  <arg value="-s" />
                  <arg value="${src-outdir}" />

                  <!-- Keep generated source files. -->
                  <arg value="-keep" />

                  <!-- Specify location of WSDL from which to generate stuff. -->
                  <arg value="${wsdl-location}" />
            </exec>

            <!--
                Delete the directory containing the generated classes and its
                contents.
            -->
            <delete dir="${gen-classdir}" />
      </target>
</project>
```

Client Main Class

The error service client main class is implemented as follows. Read the comments in the code for additional details on the error handling!

```
package com.ivan.client;

import javax.xml.ws.ProtocolException;
import javax.xml.ws.WebServiceException;

import com.ivan.service.ErrorServiceInterface;
import com.ivan.service.ErrorServiceService;
import com.ivan.service.Exception_Exception;

/**
 * This class implements a client to the Error Service.
 *
 * @author Ivan A Krizsan
 */
public class ErrorServiceClient
{
    /* Constant(s): */
    /** Will cause a system exception to be generated by the service. */
    private final static int SYS_EX_SELECTOR = 1;
    /** Will cause a service exception to be generated by the service. */
    private final static int SERVICE_EX_SELECTOR = 2;
    /** Will cause the service to produce a response, without errors. */
    private final static int NO_ERROR_SELECTOR = 3;

    public static void main(final String[] args)
    {
        try
        {
            ErrorServiceService theService = new ErrorServiceService();
            ErrorServiceInterface theProxy = theService.getErrorServicePort();

            /* Here the desired answer for the error service can be selected. */
            //int theParam = SYS_EX_SELECTOR;
            //int theParam = SERVICE_EX_SELECTOR;
            int theParam = NO_ERROR_SELECTOR;
            String theResult;

            System.out.println("Calling the Error Service with param: "
                + theParam);
            theResult = theProxy.generate(theParam);

            System.out.println("Received the result: " + theResult);
        } catch (final Exception_Exception theException)
        {
            /*
             * Note that this will only catch service exceptions, that is
             * checked exceptions that are not of the type RemoteException
             * or a subtype thereof.
             */
            System.out.println(
                "A service exception occurred calling the web service: "
                + theException.getLocalizedMessage());
        } catch (final ProtocolException theException)
        {
            /*
             * System exceptions or RemoteException occurring during remote
             * invocation of service are wrapped in ProtocolException
             * or subclass.
             * The underlying cause of the exception can be retrieved as
             * seen below.
             */
            System.out.println(
                "A system exception occurred calling the web service: "
                + theException.getLocalizedMessage());
            System.out.println(" Exception cause: " + theException.getCause());
            System.out.println(" Exception class: " + theException.getClass());
        } catch (final WebServiceException theException)
        {
            /*
             * Exceptions occurring on the client side, for instance
```

```
             * during handler invocation, are wrapped in a
             * WebServiceException (if it is not already such an exception)
             * and thrown.
             */
            System.out.println(
                "An exception occurred on the client side: "
                + theException.getLocalizedMessage());
            System.out.println(" Exception cause: " + theException.getCause());
            System.out.println(" Exception class: " + theException.getClass());
        }
    }
}
```

By selecting a different declaration of the *theParam* variable, different results can be obtained from the invocation of the error service; a system exception, a service exception and a message (no exception thrown). By undeploying the service and running the client, a client-side error can be provoked.
The behaviour of a web service client dynamically invoking a web service using the javax.xml.ws.Dispatch<T> interface is similar to the client above; if an exception is thrown on the server side that propagates out of the web service method, an exception will be thrown on the client side as well. That is, the JAX-WS runtime will translate SOAP fault messages to exceptions.

Second Error Client

If we have a client that deals with raw SOAP messages, we will not see any exceptions if an exception is thrown on the server side, instead we will see SOAP faults.

Client Main Class

The second error service client uses SAAJ to send and receive the SOAP request and response. When using SAAJ and receiving a SOAP fault message, no exception will be thrown.
SOAP faults can be detected by calling the *SOAPBody.hasFault()* method and the client can then act accordingly.

```
package com.ivan.client;

import java.net.MalformedURLException;
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

/**
 * This class implements a client of the Error Service that uses SAAJ.
 *
 * @author Ivan A Krizsan
 */
public class ErrorServiceSAAJClient
{
    /* Constant(s): */
    /** Will cause a system exception to be generated by the service. */
    private final static String SYS_EX_SELECTOR = "1";
    /** Will cause a service exception to be generated by the service. */
    private final static String SERVICE_EX_SELECTOR = "2";
    /** Will cause the service to produce a response, without errors. */
    private final static String NO_ERROR_SELECTOR = "3";
    private final static String SERVICE_URL =
        "http://localhost:8080/JAX-WSExceptionFaultHandling/ErrorServiceService";
    private final static String WSDL_NAMESPACE = "http://service.ivan.com/";


    public static void main(String[] args)
```

```
    {
        new ErrorServiceSAAJClient();
    }

    private ErrorServiceSAAJClient()
    {
        try
        {
            SOAPMessage theRequest;
            SOAPMessage theResponse;

            /*
             * Select one of the following parameters to the
             * createReqMessage method:
             * SYS_EX_SELECTOR - Service will throw a system exception.
             * SERVICE_EX_SELECTOR - Service will throw a service exception.
             * NO_ERROR_SELECTOR - Service will not throw exceptions.
             */
            theRequest = createReqMsg(SYS_EX_SELECTOR);

            theResponse = sendSOAPMessage(theRequest, SERVICE_URL);

            writeSOAPMessage(theResponse);
        } catch (final SOAPException theException)
        {
            /*
             * This kind of error will occur if:
             * - The request message could not be created.
             * - A SOAP connection could not be created.
             * - An error occurred sending the message.
             */
            System.out.println("A SOAP error occurred: "
                + theException.getLocalizedMessage());
        } catch (final MalformedURLException theException)
        {
            /*
             * This error will occur if the service URL is malformed.
             */
            System.out.println("An error occurred: "
                + theException.getLocalizedMessage());
            System.out.println("Service URL seems to be malformed: "
                + SERVICE_URL);
        }
    }

    /**
     * Creates the request message and insert the supplied parameter
     * value.
     * Note that the <generate> element belongs to the
     * http://service.ivan.com/ namespace while the <arg0> element
     * inside the <generate> element cannot have any namespace
     * specified. If it has, then the webservice will not find it and
     * act as if the parameter zero was passed.
     *
     * @param inRequestParam Parameter to enclose in request message.
     * @return A request SOAP message for the Error Service.
     * @throws SOAPException If error occurred creating SOAP message.
     */
    private SOAPMessage createReqMsg(final String inRequestParam)
        throws SOAPException
    {
        SOAPMessage theRequestMsg =
            MessageFactory.newInstance().createMessage();
        SOAPBody theBody = theRequestMsg.getSOAPBody();

        /* The <generate> element is the payload root element. */
        QName theQName = new QName(WSDL_NAMESPACE, "generate", "ik");
        SOAPElement theElem = theBody.addChildElement(theQName);

        /* The <arg0> element contains the parameter value. */
        theQName = new QName("arg0");
        theElem = theElem.addChildElement(theQName);
        theElem.setValue(inRequestParam);
        return theRequestMsg;
    }

    /**
```

```
     * Sends supplied SOAP message to the service with supplied URL.
     * Waits for response and returns it.
     *
     * @param inRequestMsg SOAP request message to send.
     * @param inServiceURL URL of service to send SOAP request message
     * to.
     * @return SOAP response message received from service.
     * @throws SOAPException If error occurs creating SOAP connection
     * or sending SOAP message.
     * @throws MalformedURLException If the service URL is malformed.
     */
    private SOAPMessage sendSOAPMessage(final SOAPMessage inRequestMsg,
        final String inServiceURL) throws SOAPException, MalformedURLException
    {
        SOAPConnection theSOAPConnection;
        URL theServiceURL;
        SOAPMessage theReplyMsg;

        /*
         * Retrieve a connection used to send the message with and
         * create an URL specifying where to send the message.
         */
        theSOAPConnection =
            SOAPConnectionFactory.newInstance().createConnection();
        theServiceURL = new URL(inServiceURL);

        theReplyMsg = theSOAPConnection.call(inRequestMsg, theServiceURL);

        return theReplyMsg;
    }

    /**
     * Writes a text-representation of the supplied SOAP message
     * to the console.
     *
     * @param inMessage SOAP message to output.
     */
    private void writeSOAPMessage(final SOAPMessage inMessage)
    {
        try
        {
            /* Output some information about the SOAP message. */
            System.out.println("\nSOAP message type: " + inMessage.getClass());
            System.out.println("SOAP message is fault: " +
                inMessage.getSOAPBody().hasFault());
            System.out.println("----------");

            inMessage.setProperty(SOAPMessage.WRITE_XML_DECLARATION, "false");
            inMessage.writeTo(System.out);
        } catch (final Exception theException)
        {
            System.out.println("An error occurred writing SOAP message: "
                + theException.getLocalizedMessage());
        }
    }
}
```

SOAP Fault Examples

As an example, we will see what the SOAP faults generated by the Error Service when a system exception and a service exception looks like. Note that GlassFish can be configured to enable or disable the capturing of stack traces. In the examples below, this feature is enabled but most part of the stack traces have been removed to conserve space.
First the SOAP fault caused by a system exception:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:Fault xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>ns2:Server</faultcode>
         <faultstring>A system exception at Thu Apr 23 21:45:18 CST 2009</faultstring>
         <detail>
            <ns2:exception class="java.lang.IllegalArgumentException" note="To disable
this feature, set com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system
property to false" xmlns:ns2="http://jax-ws.dev.java.net/">
               <message>A system exception at Thu Apr 23 21:45:18 CST 2009</message>
               <ns2:stackTrace>
                  <ns2:frame class="com.ivan.service.ErrorService"
file="ErrorService.java" line="36" method="generate"/>

                  <!-- PART OF STACKTRACE HAS BEEN REMOVED TO CONSERVE SPACE. -->

               </ns2:stackTrace>
            </ns2:exception>
         </detail>
      </ns2:Fault>
   </S:Body>
</S:Envelope>
```

Second, the SOAP fault caused by a service exception:

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:Fault xmlns:ns2="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns3="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>ns2:Server</faultcode>
         <faultstring>A service exception at Thu Apr 23 21:53:27 CST 2009</faultstring>
         <detail>
            <ns2:Exception xmlns:ns2="http://service.ivan.com/">
               <message>A service exception at Thu Apr 23 21:53:27 CST 2009</message>
            </ns2:Exception>
            <ns2:exception class="java.lang.Exception" note="To disable this feature,
set com.sun.xml.ws.fault.SOAPFaultBuilder.disableCaptureStackTrace system property to
false" xmlns:ns2="http://jax-ws.dev.java.net/">
               <message>A service exception at Thu Apr 23 21:53:27 CST 2009</message>
               <ns2:stackTrace>
                  <ns2:frame class="com.ivan.service.ErrorService"
file="ErrorService.java" line="40" method="generate"/>

                  <!-- PART OF STACKTRACE HAS BEEN REMOVED TO CONSERVE SPACE. -->

               </ns2:stackTrace>
            </ns2:exception>
         </detail>
      </ns2:Fault>
   </S:Body>
</S:Envelope>
```

10. Web Services Interoperability Technologies

Resources:
http://java.sun.com/webservices/reference/tutorials/wsit/doc/index.html

Everything in this section is basically just a reiteration of what is found in the WSIT Tutorial. It is recommended that you work through the exercises in the tutorial.

10.1 WSIT Basics

Describe WSIT, the features of each WSIT technology and the standards that WSIT Implements for each technology and how it works.

References:
WSIT Tutorial, October 2007, chapter 2.

What is WSIT?

WSIT is a joint effort of Sun and Microsoft to ensure interoperability of web services enterprise technologies. WSIT is an implementation of a number of open web services specifications to support enterprise features.
Basically this means that WSIT (Sun) and WCF (Windows Communication Foundation, Microsoft) is tested to ensure that:

- WSIT web service clients can access and consume WCF web services.
- WCF web service clients can access and consume WSIT web services.

WSIT Technologies

The technologies implemented by WSIT have the following features:

WSIT Technology	Features
Core XML	- Provides a platform agnostic, standardized, means for sharing of data. - Provides a base for all the other WSIT technologies.
Core Web Services and Optimization	- Core web services functionality. - Optimize XML used by web services to reduce application processing time and bandwidth usage.
Reliability	- Ensure delivery of application messages to webservice endpoints. Re-delivering lost messages. - Ensure that messages are delivered exactly once. - Ensure that messages are delivered in the order they were sent. Ordering messages that are out of order. - Supports session management.
Bootstrapping	- From a web service URL, retrieve the web service's WSDL, create and configure a client

	that can access and consume a web service.
Security	- Interoperable message content integrity and confidentiality, even when messages pass through intermediary nodes. - Complements transport-level security. - Supports WS-Secure Conversation, which enables the establishing of a security context when a message exchange sequence, consisting of multiple messages, is started. - Supports WS-Security Policy, which enables web services to clearly state security preferences and requirements for endpoints. - Supports WS-Trust that supports issuing, renewal, and validation of security tokens as well as trust relationship management.

WSIT Standards Implementations

WSIT includes a number of technologies which associated features listed in the following table.

WSIT Technology	Underlying Standards Implemented
Core XML	XML XML Namespaces XML Infoset XML Schema
Core Web Services and Optimization	SOAP MTOM WS-Addressing
Reliability	WS-Reliable Messaging v1.0 WS-Reliable Messaging Policy v1.0 WS-Coordination v1.0 WS-Atomic Transactions v1.0
Bootstrapping	WSDL WS-Policy v1.2 WS-Policy Attachment v1.2 WS-Metadata Exchange v1.1
Security	WS-Security Policy v1.1 WS-Security v1.1 WS-Trust v1.0 WS-Secure Conversation v1.0

How It Works

This section will give an overview how some of the WSIT technologies work. It will not describe Core XML and Core Web Services, since this is done in other parts of the document.

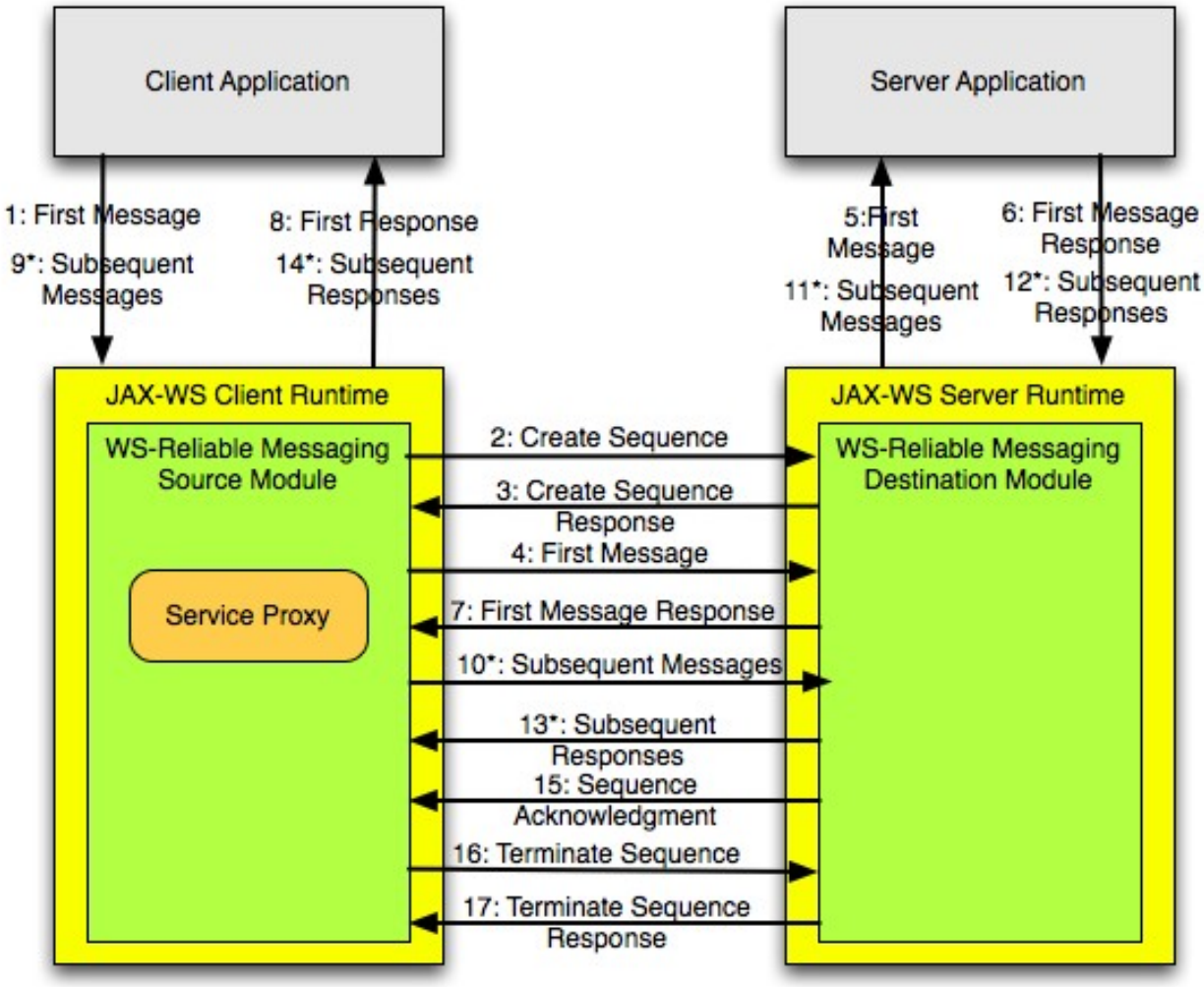
Message Optimization

The typical procedure for optimizing a SOAP message with a large binary payload is as follows:

- Identifies a SOAP message with large binary payloads.
  Usually there is some threshold size specifying when to apply message optimization.
- Removes the binary message payload from the SOAP message.
- Encodes the binary message payload, in order to reduce its size.
- Adds the binary message payload to the original SOAP message as an attachment.
- When the SOAP message has reached its destination, the binary message payload is decoded.

Reliable Messaging

WS-Reliable Messaging is implemented by adding handlers on the client and server side, which uses header blocks to transfer sequence identifier and message sequence number. The following figure shows how messages in a sequence are passed between a client and a server that uses WS-Reliable Messaging.



The above figure shows a scenario in which messages arrive in sequence and no messages are lost. The following additional scenarios exists:

- One or more messages are lost.
  The WS-Reliable Messaging Destination Module will request the WS-Reliable Messaging Source Module to resend the message(s). The source module will retain all messages of a sequence until the sequence is terminated.
- One or more messages are out of order.
  The WS-Reliable Messaging Destination Module will order the messages before they are delivered to the server application. This means that messages arriving out of order will not be delivered immediately, but retained until earlier message(s) have arrived.
- One or more duplicate messages are received.
  The WS-Reliable Messaging Destination Module discards duplicate messages.

Bootstrapping and Configuration

Bootstrapping and configuration of a web service client is done in the following manner:

- Configure the environment.
  Configure the environment in which the web service client is to run, for instance a web container.
- Retrieve the URL of a web service or a service registry.
- Retrieve information needed to build a client that can access and consume the web service. This information is usually in the form of a WSDL, made available either by the web service and/or a service registry.
- Create client artifacts.
  Using the WSDL, a client proxy and additional artifacts are created.
- Implement the web service client.

Developing Java web service clients using the WSDL-first has been discussed in detail section 4.4 of this document.

Security

Security is divided in three areas; security policy, security trust and secure conversation.

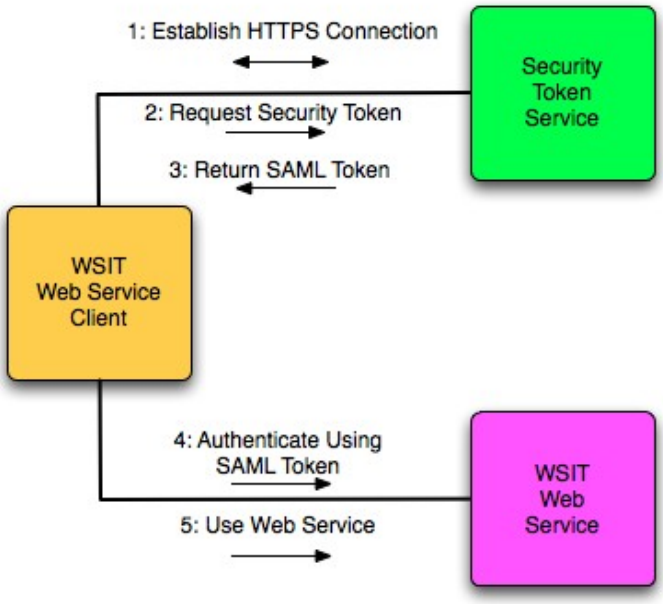
Security Policy

WSIT Web Service Security Policy builds on WS-Policy. Assertions specifying security requirements of a web service are included in the WSDL file of the service.
The following types of assertions are provided:

- Protection assertions.
  Identifies which parts of a message are to be protected.
  Specifies whether data integrity and/or data confidentiality is to be applied on the parts of a message that are to be protected.
- Conditional assertions.
  Defines conditions that must be filled by the environment in which security is to be applied.
  Defines general aspects of security.
- Security binding assertions.
  Defines the security mechanisms that are used to provide security.
  Examples:
  How data confidentiality (encryption) is to be implemented. This may include specifying which encryption algorithm to use etc.
  How data integrity (signing) is to be implemented.
- Supporting token assertions.
  Defines which types of tokens and how they may be used when applying security to parts of messages and/or individual operations of the web service.
- Web Services Security and Trust assertions.
  Define the token referencing and trust options that may be used.

Security Trust

When establishing trust between a WSIT web service and a WSIT web service client, a Security Token Service also is involved. The client needs to acquire a security token and present it to the service, before being able to use the service.



The Security Token Service also uses SOAP messages to communicate with web service clients. Establishing the HTTPS connection between the web service client and the Security Token Service (STS) is done in one of two ways:

- Username Authentication and Transport Security.
  The client presents a username token to the STS and the STS uses a certificate to authenticate to the client.
- Mutual Authentication.
  Both the client and the STS uses X509 certificates when authenticating to each other.

Secure Conversation

WS-Secure Conversation may be used when WS-Trust is not available. The establishing of a secure conversation between a WSIT web service client and a WSIT web service uses the following procedure:

- The client authenticates itself to the service by presenting its X509 certificate.
- The service authenticates itself to the client by presenting its X509 certificate.
- Client can now start using the service.

Authentication as well as the following communication uses the SOAP protocol.

Example

Somewhat extracurricular, for the curious ones, this is what a WSDL file for a WSIT web service which has WS-Reliable Messaging enabled looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://service.ivan.com/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://service.ivan.com/"
    name="CalculatorService">

    <!-- Define a WS-Policy policy regarding WS-Reliable Messaging. -->
    <ns1:Policy
        xmlns:ns1="http://schemas.xmlsoap.org/ws/2004/09/policy"
        wsu:Id="CalculatorPortBindingPolicy">
        <!--
            Standard WS-Policy normal form with an <ExactlyOne> and
            a <All> element.
        -->
        <ns1:ExactlyOne>
            <ns1:All>
                <!-- Enable WS-Reliable Messaging. -->
                <ns3:RMAssertion
                    xmlns:ns3="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"/>
                <!-- Enable WS-Reliable Messaging flow control. -->
                <ns2:RmFlowControl
                    xmlns:ns2="http://schemas.microsoft.com/net/2005/02/rm/policy"/>
                <!-- Enable WS-Reliable Messaging message ordering. -->
                <ns4:Ordered xmlns:ns4="http://sun.com/2006/03/rm"/>
                <!-- Optionally use WS-Addressing. -->
                <ns5:UsingAddressing
                    xmlns:ns5="http://www.w3.org/2006/05/addressing/wsdl"
                    ns1:Optional="true"/>
            </ns1:All>
        </ns1:ExactlyOne>
    </ns1:Policy>

    <types>
        <xsd:schema>
            <xsd:import namespace="http://service.ivan.com/"
```

```
schemaLocation="http://localhost:8080/WSIT-CalculatorService/CalculatorService?
xsd=1"></xsd:import>
        </xsd:schema>
    </types>

    <message name="add">
        <part name="parameters" element="tns:add"></part>
    </message>
    <message name="addResponse">
        <part name="parameters" element="tns:addResponse"></part>
    </message>

    <portType name="Calculator">
        <operation name="add">
            <input message="tns:add"></input>
            <output message="tns:addResponse"></output>
        </operation>
    </portType>

    <binding name="CalculatorPortBinding" type="tns:Calculator">
        <!--
            Apply the above defined policy regarding
            WS-Reliable Messaging to this binding.
        -->
        <ns6:PolicyReference
            xmlns:ns6="http://schemas.xmlsoap.org/ws/2004/09/policy"
            URI="#CalculatorPortBindingPolicy"/>
        <soap:binding
            transport="http://schemas.xmlsoap.org/soap/http"
            style="document"/>
        <operation name="add">
            <soap:operation soapAction="add"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>

    <service name="CalculatorService">
        <port name="CalculatorPort" binding="tns:CalculatorPortBinding">
            <soap:address location="TBA"/>
        </port>
    </service>
</definitions>
```

Here is what a SOAP request message sent to the web service looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header>
        <!--
            First a number of header blocks related to WS-Addressing.
        -->
        <To xmlns="http://www.w3.org/2005/08/addressing">
            http://localhost:8080/WSIT-CalculatorService/CalculatorService
        </To>
        <Action xmlns="http://www.w3.org/2005/08/addressing">add</Action>
        <ReplyTo xmlns="http://www.w3.org/2005/08/addressing">
            <Address>http://www.w3.org/2005/08/addressing/anonymous</Address>
        </ReplyTo>
        <MessageID xmlns="http://www.w3.org/2005/08/addressing">
            uuid:4cd797d2-b4d0-4a73-8bd1-0d9f5c70cc1b
        </MessageID>

        <!--
            Here are the WS-Reliable Messaging header blocks.
        -->
        <ns2:Sequence
            xmlns:ns2="http://schemas.xmlsoap.org/ws/2005/02/rm"
            xmlns:ns3="http://www.w3.org/2005/08/addressing"
            xmlns:ns4="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
            xmlns:ns5="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
```

```
wssecurity-utility-1.0.xsd"
            xmlns:ns6="http://schemas.microsoft.com/ws/2006/05/rm">
            <ns2:Identifier>uuid:c136da8c-7d88-475f-8e99-b9733c896a12</ns2:Identifier>
            <ns2:MessageNumber>1</ns2:MessageNumber>
        </ns2:Sequence>

        <ns2:AckRequested
            xmlns:ns2="http://schemas.xmlsoap.org/ws/2005/02/rm"
            xmlns:ns3="http://www.w3.org/2005/08/addressing"
            xmlns:ns4="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
            xmlns:ns5="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-utility-1.0.xsd"
            xmlns:ns6="http://schemas.microsoft.com/ws/2006/05/rm">
            <ns2:Identifier>
                uuid:c136da8c-7d88-475f-8e99-b9733c896a12
            </ns2:Identifier>
        </ns2:AckRequested>
    </S:Header>

    <!--
        This is the SOAP body containing the request to the service to add
        the numbers 1 and 2.
    -->
    <S:Body>
        <ns2:add xmlns:ns2="http://service.ivan.com/">
            <inNumber1>1</inNumber1>
            <inNumber2>2</inNumber2>
        </ns2:add>
    </S:Body>
</S:Envelope>
```

The response to the above request looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header>
        <!--
            The response only contains WS-Addressing header blocks.
        -->
        <To xmlns="http://www.w3.org/2005/08/addressing">
            http://www.w3.org/2005/08/addressing/anonymous
        </To>
        <Action xmlns="http://www.w3.org/2005/08/addressing">
            http://service.ivan.com/Calculator/addResponse
        </Action>
        <MessageID xmlns="http://www.w3.org/2005/08/addressing">
            uuid:709d3e3e-ba37-4a51-88d1-2a2a505a739d
        </MessageID>
        <RelatesTo xmlns="http://www.w3.org/2005/08/addressing">
            uuid:4cd797d2-b4d0-4a73-8bd1-0d9f5c70cc1b
        </RelatesTo>
    </S:Header>

    <!--
        SOAP body containing the response message from the service.
        The answer seems to be correct: 1 + 2 = 3
    -->
    <S:Body>
        <ns2:addResponse xmlns:ns2="http://service.ivan.com/">
            <return>3</return>
        </ns2:addResponse>
    </S:Body>
</S:Envelope>
```

The above request and response messages were generated by the GlassFish web service test tool.

10.2 WSIT Clients

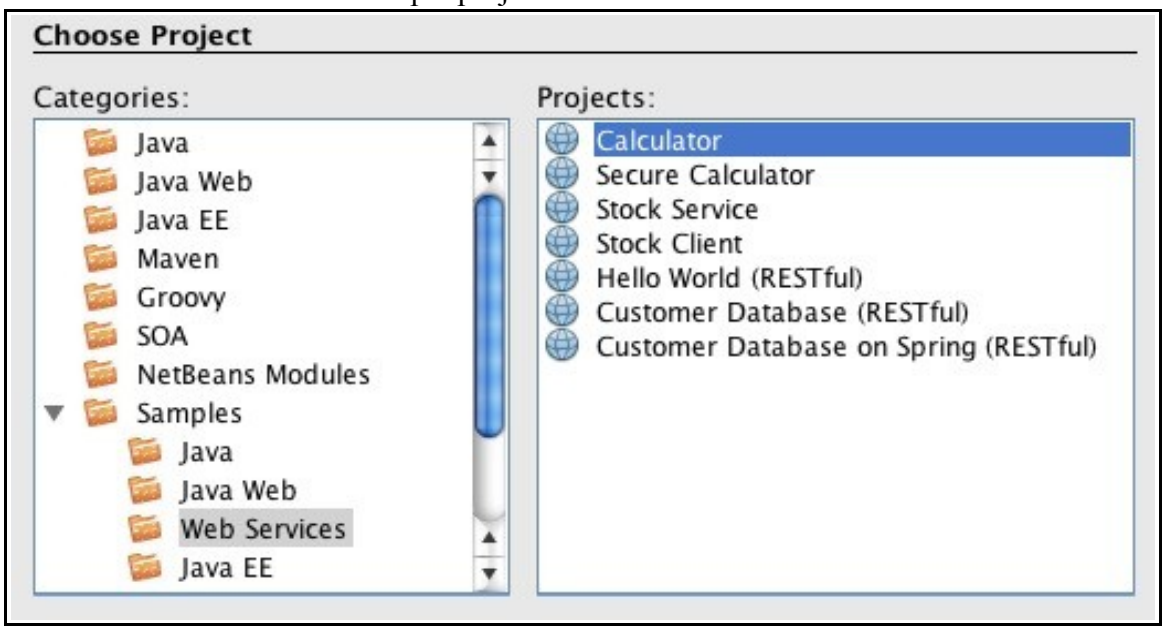
Describe how to create a WSIT client from a Web Service Description Language (WSDL) file.

References:
WSIT Tutorial, October 2007, chapter 3 and 4.

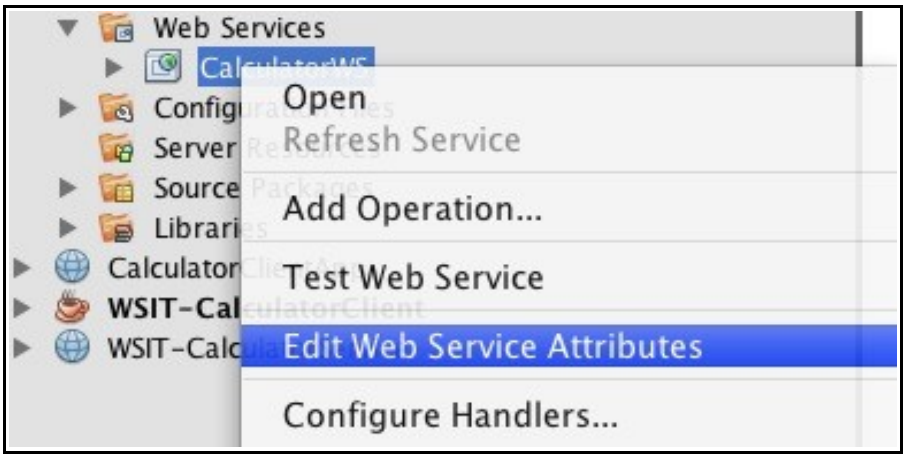
Creating a WSIT Web Service

In order to be able to actually try the WSIT client we are to created, let us first create a WSIT web service that we'll also configure to use WS-Reliable Messaging. To make things easy for me, I will, for once, depart from the road of Eclipse development and use NetBeans 6 to develop the WSIT web service. In fact, NetBeans contains a Calculator web service example that will be used, so there won't be much developing done.

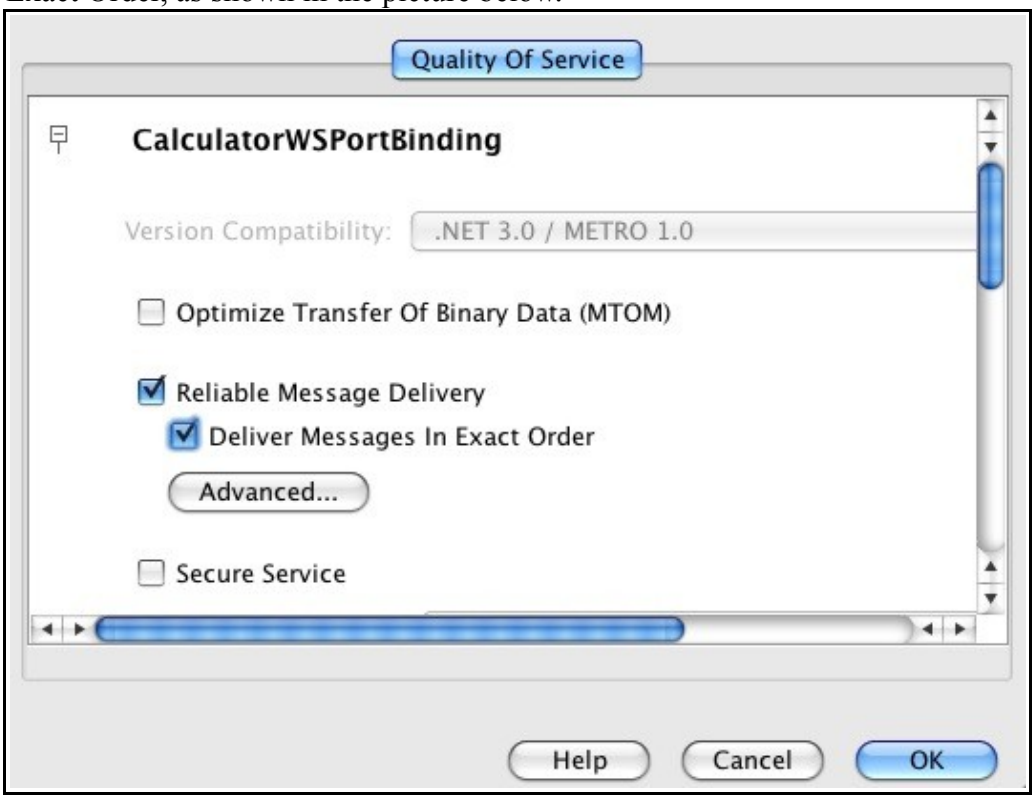
- In NetBeans, go to the File menu and select New Project...
- Select the web services example project Calculator.



- In the CalculatorApp project, open the Web Services node and right-click on the CalculatorWS web service and select Edit Web Service Attributes.



- In the Quality Of Service settings, enable Reliable Message Delivery and Deliver Messages In Exact Order, as shown in the picture below.



- Build the web service and deploy it to GlassFish.
- Retrieve the Calculator web service's WSDL URL, which will be used when developing the WSIT client in the next section.

Developing the WSIT Client

With the WSIT web service in place, we are now ready to develop the client. Developing a WSIT client in Java is almost identical to the WSDL-first development approach that we have seen earlier. Before we can start developing the client, we need to obtain the following Metro libraries that we will need:

- webservices-api.jar
- webservices-extra-api.jar
- webservices-extra.jar
- webservices-rt.jar
- webservices-tools.jar

Download them from the Metro home or extract them from the NetBeans 6 distribution.

Now we can start developing the WSIT client, which will be done in Eclipse.

- Create a Java project in Eclipse.
  I name my project WSIT-CalculatorClient.
- In the root of the project, create a lib folder.
- Copy all the above Metro library JAR files to the lib folder.
- Add all the Metro libraries to the build path of the project.
- In the root of the project, create a folder named wsimport_generated.

- In the root of the project, create a file named build.xml and insert the following contents into the file:

```
<?xml version="1.0"?>

<project basedir=".">

      <property name="class-dir" value="${basedir}/bin" />
      <property name="wsimport-outdir" value="${basedir}/wsimport_generated" />
      <property name="gen-classdir" value="${wsimport-outdir}/com/" />
      <property name="src-outdir" value="${basedir}/src/" />
      <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport" />
      <property name="wsdl-location" value="http://localhost:8080/WSIT-
CalculatorService/CalculatorService?wsd" />

      <!--
            Generates the client artifacts from the WSDL of the
            WSIT Calculator service.
      -->
      <target name="generate-client">
            <exec executable="${wsimport-cmd}">
                  <arg value="-verbose" />

                  <!-- Specify where to write other generated files. -->
                  <arg value="-d" />
                  <arg value="${wsimport-outdir}" />

                  <!-- Specify where to write generated source files. -->
                  <arg value="-s" />
                  <arg value="${src-outdir}" />

                  <!-- Keep generated source files. -->
                  <arg value="-keep" />

                  <!-- Specify location of WSDL from which to generate stuff. -->
                  <arg value="${wsdl-location}" />
            </exec>

            <!--
           Delete the directory containing the generated classes and its
           contents.
        -->
            <delete dir="${gen-classdir}" />
      </target>
</project>
```

- Modify the following properties in the above build script to match your environment:
  wsimport-cmd – Location of the wsimport command.
  wsdl-location – URL of the WSDL document of the web service the client is to access.
- Run the Ant build file.
  This will generate a number of classes in the *org.me.calculator* package.
  As before, these are the JAXB classes, the service class and the port class for the Calculator web service.
- Implement the main class of the client.

```
package org.me.calculator;

import com.sun.xml.ws.Closeable;

public class CalculatorClientMain
{
    private final static int[][] NUMBERS =
    {
        {1, 3}, {5, 3}, {6, 19}
    };

    public static void main(String[] args)
    {
        try
        {
            CalculatorWSService theCalcService = new CalculatorWSService();
```

```
            CalculatorWS theCalcPort = theCalcService.getCalculatorWSPort();

            for (int i = 0; i < NUMBERS.length; i++)
            {
                int theNumber1 = NUMBERS[i][0];
                int theNumber2 = NUMBERS[i][1];

                int theResult = theCalcPort.add(theNumber1, theNumber2);

                System.out.println("\n\n" + theNumber1 + " + " + theNumber2
                    + " = " + theResult);
            }

            /*
             * Need to close the port after we are done using it, since
             * we use WS-Reliable Messaging and we need to tell it that
             * it no longer have to retain messages for this session.
             */
            ((Closeable)theCalcPort).close();
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }
}
```

- The client can now be run and should produce a number of sums.

Note that the Metro libraries take care of WS-Reliable Messaging for us on the client side – there are no special measures needed on our account, apart from adding the libraries to the project.
As an exercise, I leave to the reader to add SOAP message logging to the client, in order to verify that the SOAP messages indeed contain WS-Reliable Messaging header blocks.

10.3 Message Optimization

Describe how to configure web service providers and clients to use message optimization.

References:
WSIT Tutorial, October 2007, chapter 5.
http://www.w3.org/Submission/WS-MTOMPolicy/
http://www.w3.org/TR/soap12-mtom/

In this section, we'll develop a web service that enables us to retrieve and upload small pictures as well as a web service client that retrieves pictures.
In addition to the tools we've used before, I will introduce soapUI, which is an excellent tool for testing web services. There is even a free version which you can download from this webpage:
http://www.soapui.org/

Developing the Picture Web Service

- Create a Dynamic Web project with the name MTOMWebService in Eclipse.
- Create the source code package *com.ivan.service*.
- Create a directory named numbers in the WebContent folder of the project.
- Copy ten image files, named 0.jpg, 1.jpg etc. up to 9.jpg, into the numbers directory created in the previous step. I use pictures of the digits 0 to 9 to be able to easily verify that a request yields the correct picture.
- Implement the web service.
  Note that using the @MTOM annotation is the recommended way to enable/disable MTOM in the endpoint implementation class. Do not use the @BindingType annotation with the parameters *SOAPBinding.SOAP11HTTP MTOM BINDING* or *SOAPBinding.SOAP11HTTP MTOM BINDING*.

```
package com.ivan.service;

import java.awt.Image;
import java.io.IOException;
import java.io.InputStream;
import java.util.Date;

import javax.annotation.Resource;
import javax.imageio.ImageIO;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.servlet.ServletContext;
import javax.xml.ws.WebServiceContext;
import javax.xml.ws.handler.MessageContext;
import javax.xml.ws.soap.MTOM;

/**
 * This class implements a web service that enables clients to retrieve
 * pictures.
 * Enabling MTOM on the server side is done simply by adding the
 * @MTOM annotation, in which one can specify the enabled/disabled status
 * and a threshold above which message optimization is to be applied.
 * See JAX-WS 2.1 Specification, section 7.14.2 and 6.5.2.
 */
@WebService()
@MTOM(enabled=true, threshold=512)
public class PictureManager
{
    @Resource
    private WebServiceContext mWSContext;

    /**
```

```
     * Retrieves the picture with the supplied number.
     *
     * @parm inImageNumber Number of image to retrieve.
     * @return Image with number as specified, or null if no such
     * no such image exits.
     */
    @WebMethod(operationName = "retrievePicture")
    public Image retrievePicture(@WebParam(name = "inImageNumber")
            final int inImageNumber)
    {
        Image theImage = null;

        /*
         * In order to be able to retrieve the image the client wants
         * we need to get the servlet context.
         * Using the servlet context, we can then retrieve the proper
         * image located in the web contents of this application.
         */
        MessageContext theMsgContext = mWSContext.getMessageContext();
        ServletContext theServletContext = (ServletContext) theMsgContext.get(
                MessageContext.SERVLET_CONTEXT);

        if (theServletContext != null)
        {
            InputStream thePicStream;

            /* Try reading the image. */
            thePicStream = theServletContext.getResourceAsStream(
                    "numbers/" + inImageNumber + ".jpg");
            try
            {
                theImage = ImageIO.read(thePicStream);
            } catch (IOException theException)
            {
                System.out.println("An error occurred trying to read image:");
                theException.printStackTrace();
            }
        } else
        {
            System.out.println("No servlet context, cannot retrieve image!");
        }

        return theImage;
    }

    /**
     * Accepts an image and returns a message confirming having
     * received the image.
     *
     * @parm inImageNumber Number of image to retrieve.
     * @return Image with number as specified, or null if no such
     * no such image exits.
     */
    @WebMethod(operationName = "supplyPicture")
    public String supplyPicture(@WebParam(name = "inImage")
        final Image inImage)
    {
        String theMessage =
            "PictureManager received a picture with height " +
            inImage.getHeight(null) + " and width " +
            inImage.getWidth(null) + " at " + new Date();

        return theMessage;
    }
}
```

- The web service is now ready and can be deployed to GlassFish.

Testing the Web Service

Now we will test the web service we just developed, using first the GlassFish web service testing tool, which should be familiar by now, and then soapUI. Note that we will only be able to test the *retrievePicture* function of the web service, since I do not know how to insert picture data into messages in these tools.

Testing in GlassFish

If we test the web service using the GlassFish web service test tool, we can see the following message exchange taking place when the parameter 4 was entered:

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Header/>
    <S:Body>
        <ns2:retrievePicture xmlns:ns2="http://service.ivan.com/">
            <inImageNumber>4</inImageNumber>
        </ns2:retrievePicture>
    </S:Body>
</S:Envelope>
```

Request SOAP message sent to the picture retrieving web service.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:retrievePictureResponse xmlns:ns2="http://service.ivan.com/">
            <return>iVBORw0KGgoAAAANSUhE...</return>
        </ns2:retrievePictureResponse>
    </S:Body>
</S:Envelope>
```

Response SOAP message received from the picture retrieving web service.
Data in the <return> element has been shortened.

MTOM will not be activated when using the GlassFish web service testing tool, so the above SOAP messages does not contain any MTOM related information.
To ensure that the web service has MTOM enabled, take a look at the WSDL of the service:

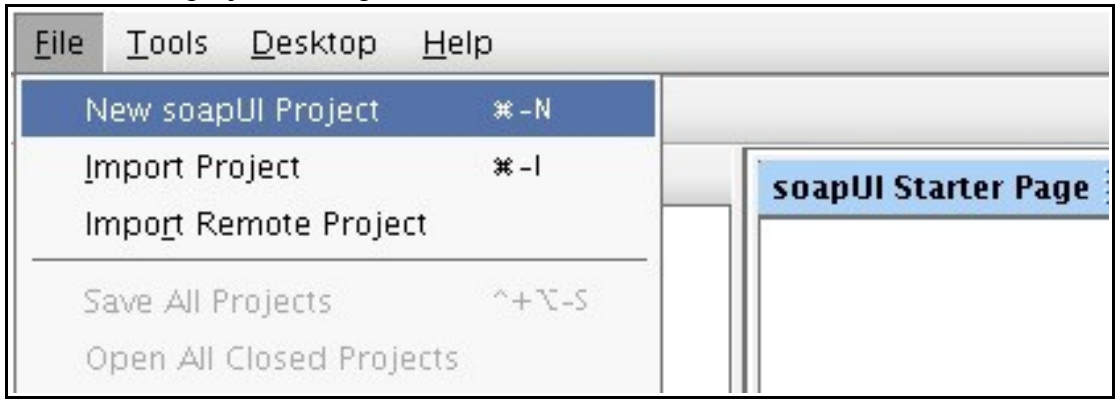
```
...
<definitions
      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:tns="http://service.ivan.com/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      targetNamespace="http://service.ivan.com/"
      name="PictureManagerService">

      <!-- Here the policy declaring that MTOM is to be used is defined. -->
    <!-- ************************************************************ -->
      <ns1:Policy xmlns:ns1="http://www.w3.org/ns/ws-policy"
             wsu:Id="PictureManagerPortBinding_MTOM_Policy">
          <ns1:ExactlyOne>
              <ns1:All>
                  <ns2:OptimizedMimeSerialization
                      xmlns:ns2="http://schemas.xmlsoap.org/ws/2004/09/policy/optimizedm
imeserialization">
                   </ns2:OptimizedMimeSerialization>
              </ns1:All>
          </ns1:ExactlyOne>
      </ns1:Policy>

    ...
    <binding name="PictureManagerPortBinding" type="tns:PictureManager">
        <!-- This binding uses the MTOM policy declared above. -->
        <!-- ********************************************** -->
        <ns3:PolicyReference xmlns:ns3="http://www.w3.org/ns/ws-policy"
            URI="#PictureManagerPortBinding_MTOM_Policy"></ns3:PolicyReference>
```

```
            <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document"></soap:binding>
            <operation name="retrievePicture">
                  <soap:operation soapAction=""></soap:operation>
                  <input>
                        <soap:body use="literal"></soap:body>
                  </input>
                  <output>
                        <soap:body use="literal"></soap:body>
                  </output>
            </operation>
            <operation name="supplyPicture">
                  <soap:operation soapAction=""></soap:operation>
                  <input>
                        <soap:body use="literal"></soap:body>
                  </input>
                  <output>
                        <soap:body use="literal"></soap:body>
                  </output>
            </operation>
      </binding>
    ...
</definitions>
```

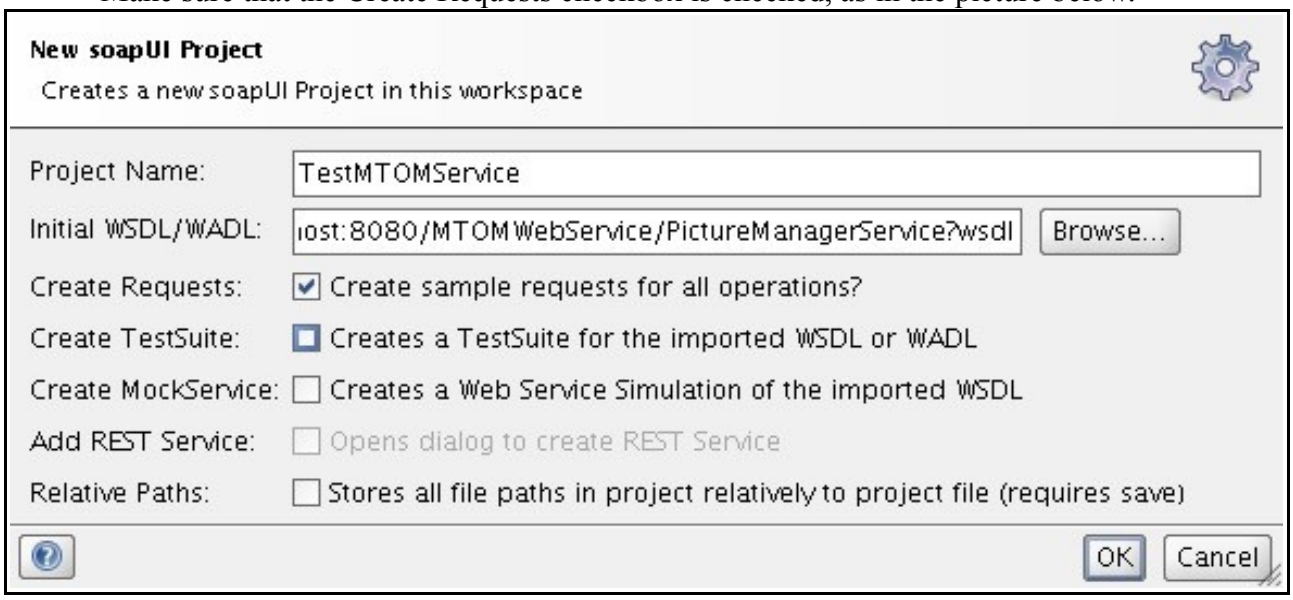
Testing with soapUI

This document will only give a very brief introduction on how to test a web service using soapUI. Please refer to the soapUI documentation for further details. It is assumed that the standalone version of soapUI has been downloaded and successfully installed.

- Obtain the URL of the WSDL document of the web service.
  In my case, it is:
  http://localhost:8080/MTOMWebService/PictureManagerService?wsdl
- Launch soapUI.
- Create a new project in soapUI:



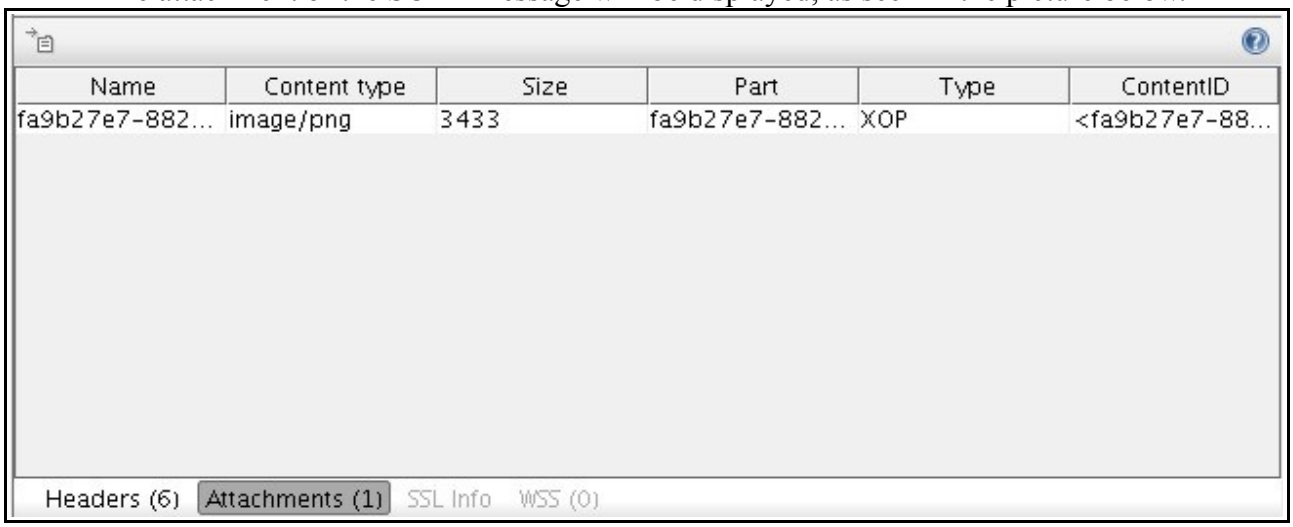
- In the configuration dialog for the new project, enter a project name and the URL of the WSDL of the web service to test.
  Make sure that the Create Requests checkbox is checked, as in the picture below.



- Locate the Request 1 node in the Navigator window on the left and double click it.
  A window showing the SOAP request message should open, as seen in the picture below.

- Modify the contents of the <inImageNumber> element in the Request 1 window, replacing the content with a digit 0 – 9.
- Click the little green triangle in the upper left corner of the Request 1 window.
  This sends the SOAP request to the web service, which should answer with a SOAP response message that appears to the right of the request in the Request 1 window.

- Click the Attachments button below the response message.
  The attachment of the SOAP message will be displayed, as seen in the picture below.



- Examining the SOAP response message, we see that the <return> element does not contain any data, like in the previous test, but an <Include> element.
  The data has been extracted, compressed and attached to the SOAP message as an attachment by the MTOM mechanism.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:retrievePictureResponse xmlns:ns2="http://service.ivan.com/">
         <return>
            <Include href="cid:fa9b27e7-8821-4448-87a7-
e5ab5acf655b@example.jaxws.sun.com" xmlns="http://www.w3.org/2004/08/xop/include"/>
         </return>
      </ns2:retrievePictureResponse>
   </S:Body>
</S:Envelope>
```

Developing the Picture Web Service Client

In this section we'll develop a client to the above MTOM web service. We will also see how to enable or disable MTOM programmatically on the client side.

- Create a Java project in Eclipse for the client.
  My project is named MTOMWebServiceClient.
- Create a director named wsimport_generated in the root of the project.
- Create a file named build.xml in the root of the project, with the following contents:

```
<?xml version="1.0"?>

<project basedir=".">

    <property name="class-dir" value="${basedir}/bin" />
    <property name="wsimport-outdir" value="${basedir}/wsimport_generated" />
    <property name="gen-classdir" value="${wsimport-outdir}/com/" />
    <property name="src-outdir" value="${basedir}/src/" />
    <property name="wsimport-cmd"
value="/System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/wsimport" />
    <property name="wsdl-location"
value="http://localhost:8080/MTOMWebService/PictureManagerService?wsdl" />

    <!--
        Generates the client artifacts from the WSDL for the
        WSIT MTOM Picture Retrieval client.
    -->
    <target name="generate-client">
        <exec executable="${wsimport-cmd}">
            <arg value="-verbose" />

            <!-- Specify where to write other generated files. -->
            <arg value="-d" />
            <arg value="${wsimport-outdir}" />

            <!-- Specify where to write generated source files. -->
            <arg value="-s" />
            <arg value="${src-outdir}" />

            <!-- Keep generated source files. -->
            <arg value="-keep" />

            <!-- Specify location of WSDL from which to generate stuff. -->
            <arg value="${wsdl-location}" />
        </exec>

        <!--
            Delete the directory containing the generated classes and its
            contents.
        -->
        <delete dir="${gen-classdir}" />
    </target>

</project>
```

- Modify the following properties in the above build script to match your environment:
  wsimport-cmd – Location of the wsimport command.
  wsdl-location – URL of the WSDL used to generate client artifacts.
- Run the above Ant script and generate client artifacts.
- Create a directory named lib in the root of the project.
- Copy all the Metro library JAR files to the lib folder.
  See above concerning which Metro JAR files to include and how to obtain them.
- Add all the Metro libraries to the build path of the project.

- Implement the main class of the client.
  As in the comments, the following method of trying to disable MTOM on the client side does not work and should not be used:
  SOAPBinding theSOAPBinding = (SOAPBinding)((BindingProvider)thePort).getBinding();
  theSOAPBinding.setMTOMEnabled(false);

```
package com.ivan.client;

import java.awt.image.BufferedImage;
import java.io.ByteArrayInputStream;
import java.io.IOException;

import javax.imageio.ImageIO;
import javax.imageio.stream.ImageInputStream;
import javax.swing.JFrame;
import javax.xml.ws.soap.MTOMFeature;

import com.ivan.service.PictureManager;
import com.ivan.service.PictureManagerService;

/**
 * This class implements a client to the picture manager service.
 * In order for MTOM to work in this, standalone JavaSE application,
 * the METRO web service stack JAR files must be present on the Java
 * build path.
 */
public class PictureServiceClientMain
{
    public static void main(String[] args)
    {
        try
        {
            PictureManagerService theService;
            PictureManager thePort;

            /* Create a new instance of the picture manager service. */
            theService = new PictureManagerService();

            /*
             * When retrieving the port, enable MTOM on the client side.
             * Note that this only affects data sent by the client to
             * the web service, for instance when sending pictures to
             * the service.
             * The following method does not work to disable MTOM:
             * SOAPBinding theSOAPBinding =
             * (SOAPBinding)((BindingProvider)thePort).getBinding();
             * theSOAPBinding.setMTOMEnabled(false);
             */
            thePort = theService.getPictureManagerPort(new MTOMFeature(true));

            /*
             * Request an image from the web service.
             * Valid image numbers are zero to nine.
             */
            int theImageNumber = 5;
            byte[] theResultBytes = thePort.retrievePicture(theImageNumber);

            System.out.println("Done retrieving image. Length: "
                + theResultBytes.length);

            showPicture(theResultBytes);

            /* Send the image we just received back to the web service. */
            String theMessage = thePort.supplyPicture(theResultBytes);

            System.out.println("Service response: " + theMessage);
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    }

    /**
     * Shows the picture, which data is supplied, in a window.
```

```
     *
     * @param inImageBytes Picture data.
     * @throws IOException If error occurs reading picture data.
     */
    private static void showPicture(byte[] inImageBytes) throws IOException
    {
        ByteArrayInputStream theBIS = new ByteArrayInputStream(inImageBytes);
        ImageInputStream theIIS = ImageIO.createImageInputStream(theBIS);
        BufferedImage theImg = ImageIO.read(theIIS);

        JFrame theWindow = new JFrame();
        theWindow.setSize(150, 150);
        theWindow.setVisible(true);
        theWindow.getGraphics().drawImage(theImg, 50, 50, null);
    }
}
```

- The client can now be run.
  A small window with a picture in it should appear and the console will contain output similar to the following:

```
Done retrieving image. Length: 3463
Service response: PictureManager received a picture with height 58 and
width 41 at Tue May 05 22:58:09 CST 2009
```

- As an exercise for the reader, enable and disable MTOM on the client side and use a package capturing tool to inspect the SOAP messages sent between the client and the server. For Macintosh OS X, I can recommend the open source, free, network protocol analyzer Packet Peeper. For Windows and Linux etc, there is Wireshark.

10.4 WCF Web Service Clients

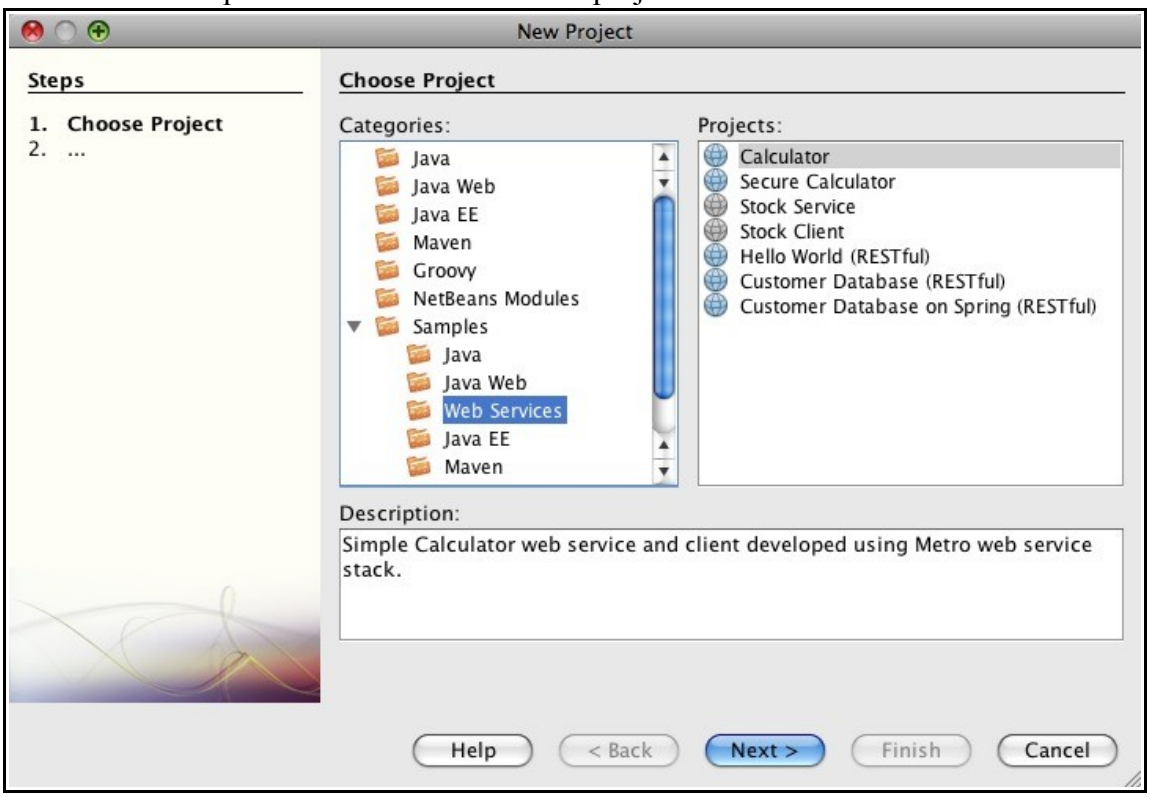
Create a Microsoft Windows Communication Foundation (WCF) client that accesses a Java web service.

References:
WSIT Tutorial, October 2007, chapter 9 and 10.

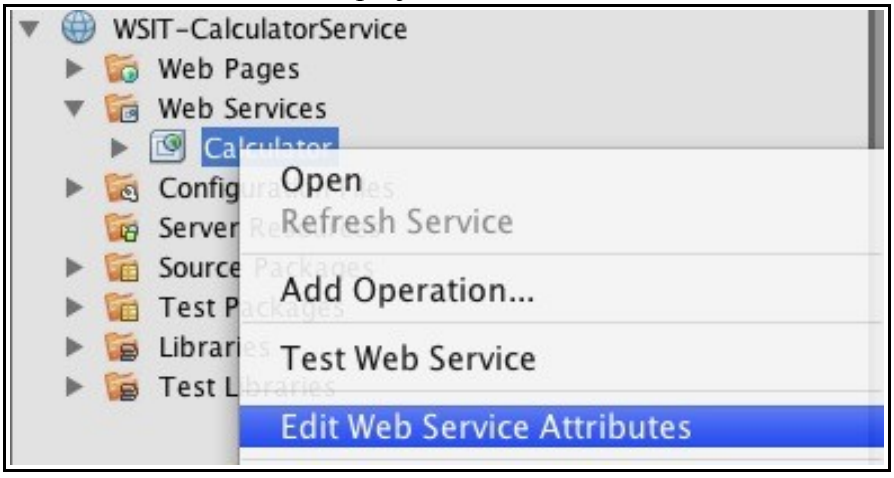
Creating and Configuring the Java Web Service

To create and configure the web service that will be accessed by a WCF client, NetBeans will be used, since it provides better support in this area.

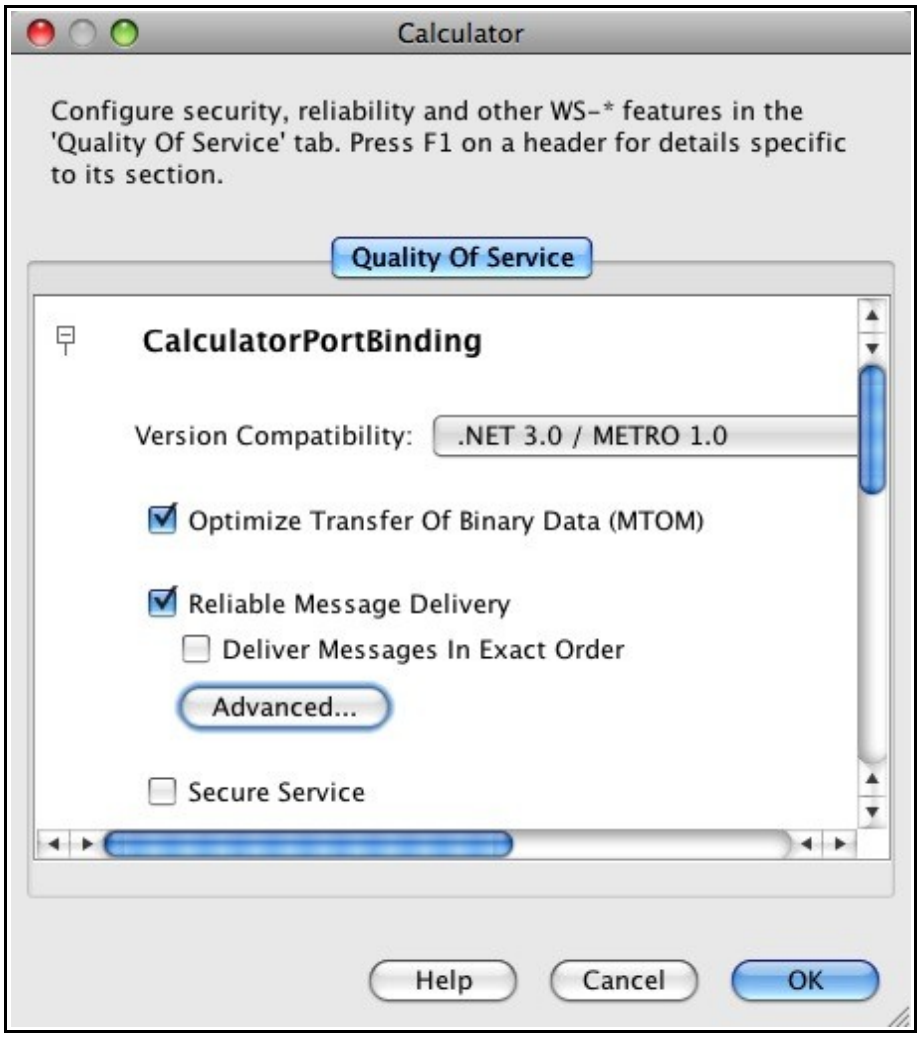
- Create the Sample Web Services Calculator project.



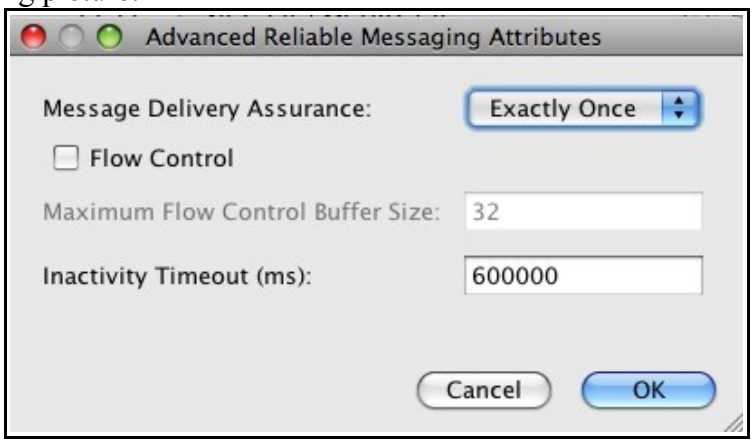
- Open the Web Services node in the project and select Edit Web Service Attributes.



- Configure the web service attributes according to the picture below.
  Note that it is important to set .NET 3.0 / METRO 1.0 compatibility!
  Also, do not enable Deliver Messages In Exact Order, since this feature does not seem to be supported in .NET 3.0.



- Click the Advanced button in the above dialog and configure Reliable Messaging according to the following picture:

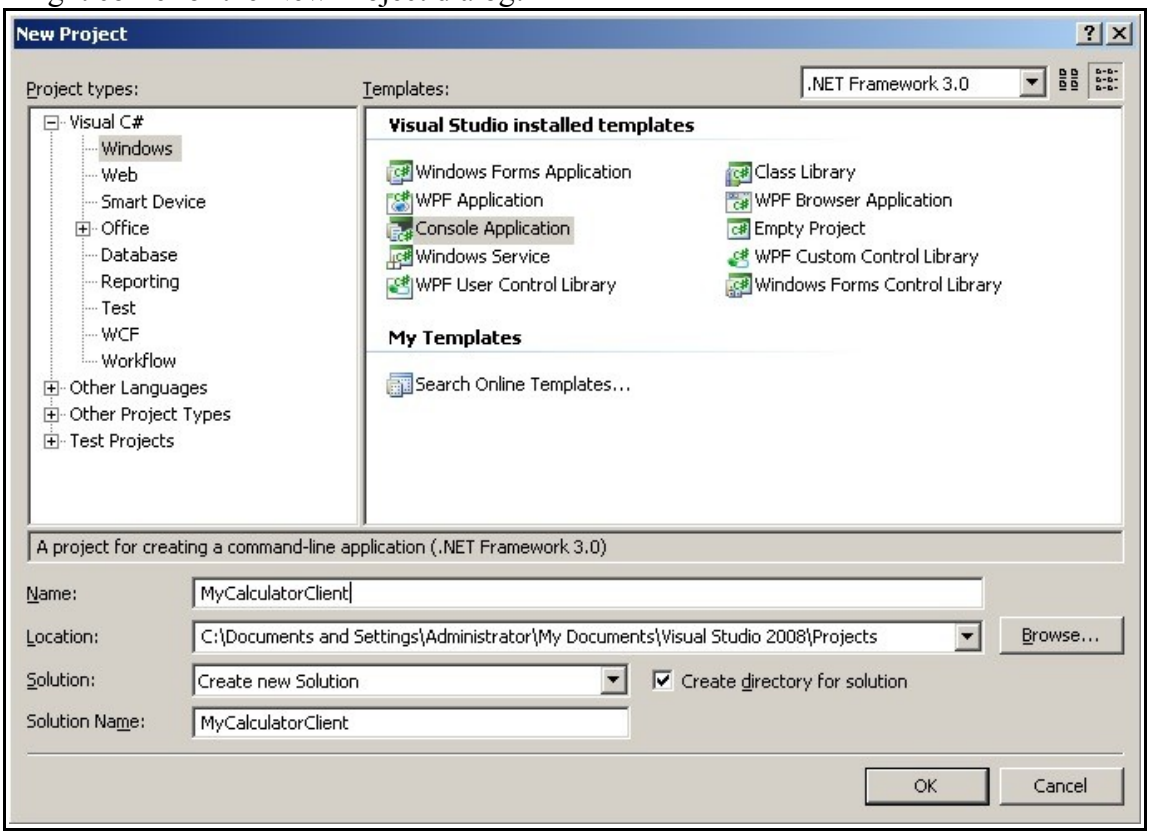


- Build and deploy the web service to GlassFish.

Creating the WCF Web Service Client

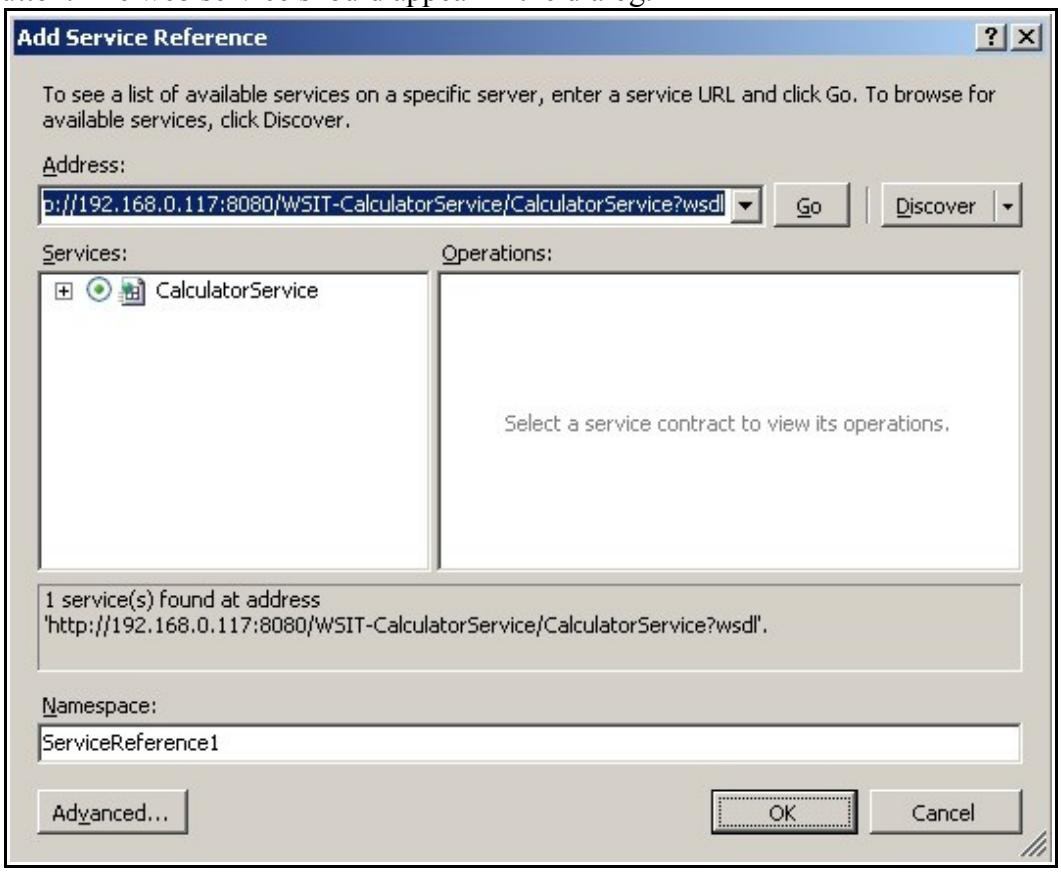
To develop the WCF web service client, Visual Studio 2008 Professional will be used. A 90-day trial version can be downloaded from Microsoft.

- Find the URL of the Calculator web service's WSDL document.
  In my case, it is:
  http://192.168.0.117:8080/WSIT-CalculatorService/CalculatorService?wsdl
- Open Visual Studio.
- Create a new Console Application project called MyCalculatorClient.
  Very important! We must use the .NET Framework 3.0 – verify the setting in the upper right corner of the New Project dialog!

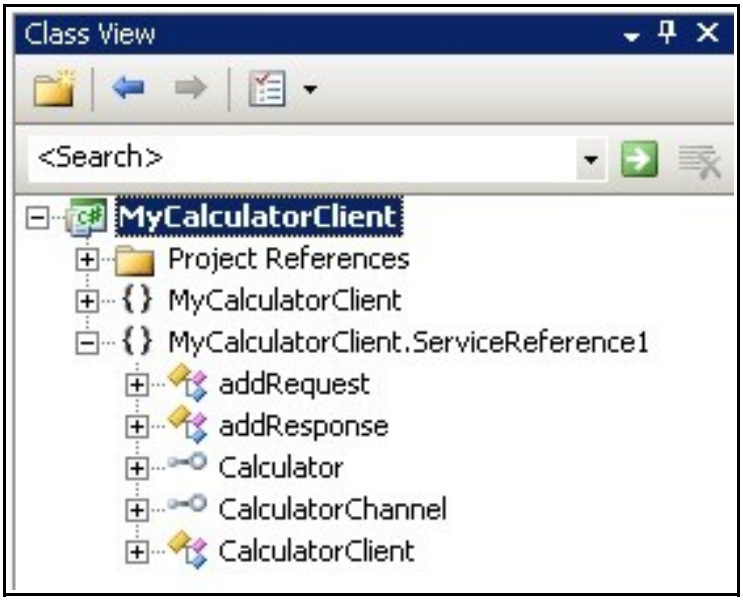


- Go to the Project menu and select Add Service Reference.

- In the dialog that appears, paste the address of the web service WSDL and click the Go button. The web service should appear in the dialog.



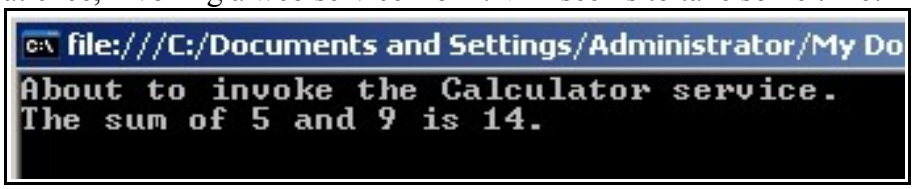
- Click the OK button in the Add Service Reference dialog.
- In the Class View, expand the project node and the MyCalculatorClient.ServiceReference1 node.
  The class we are to use to access the web service is named *CalculatorClient*.



- In the window displaying the class *Program*, implement the client:

```
using System;
using System.Collections.Generic;
using System.Text;
using MyCalculatorClient.ServiceReference1;

namespace MyCalculatorClient
{
    class Program
    {
        static void Main(string[] args)
        {
            int theNum1 = 5;
            int theNum2 = 9;
            int theSum;

            Console.WriteLine("About to invoke the Calculator service.");
            CalculatorClient theClient = new CalculatorClient();
            theSum = theClient.add(theNum1, theNum2);

            Console.WriteLine("The sum of {0} and {1} is {2}.", theNum1, theNum2,
theSum);

            theClient.Close();

            // This is to avoid the console window closing when the program ends.
            while (1 < 2)
            {
            }
        }
    }
}
```

- Go to the Debug menu and select Start Without Debugging.
  A console window appears and the client prints the result of the web service invocation.
  Have patience, invoking a web service from .NET seems to take some time!



Alternative Approach

Chapter 10 of the WSIT Tutorial describes how to use a tool called svcutil to generate WCF client artifacts. I see no use in reiterating the information here, instead I urge curious readers to refer to the chapter in question.

10.5 WCF and Java Web Service Interoperability

Describes the best practices for production and consumption of data interoperability between WCF web services and Java web service clients or between Java web services and WCF web service clients.

References:
WSIT Tutorial, October 2007, chapter 11.

Clients/services written in Java (WSIT) uses JAXB to perform the following tasks:

- XML serialization.
- Generating XML schemas from Java classes.
- Generating Java classes from XML schemas.

WCF uses *DataContractSerializer* or *XmlSerializer* in .NET to perform similar tasks. The difference between the two mechanisms are:

- JAXB 2.0 supports all of XML schema, while corresponding WCF mechanisms supports different XML schema sets.
- The mapping of programming language data types to XML schema constructs differ between the two technologies.

The type of serializer can be chosen when invoking the svcutil program to generate client artifacts:

```
svcutil.exe /serializer:DataContractSerializer MyService.wsdl
```

The result is that special care must be taken when developing web service, especially when using the Java first development approach.

Web Service Java First

The following recommendations are to be taken into consideration when developing a web service, using the Java first approach. For examples, please refer to the WSIT Tutorial, October 2007, chapter 11.

- Java primitives, such as *long*, *int* etc, maps to one kind of XML schema representation while corresponding Java wrapper classes, such as *Long*, *Integer* etc, maps to a different XML schema representation. This will cause the .NET bindings to vary accordingly.
- The Java class *BigDecimal* maps to the XML schema type *decimal*. .NET in turn maps the XML schema type to *System.decimal*. *BigDecimal* supports arbitrary precision but *System.decimal* does not, so care must be taken as to, in the Java code, use numbers in the range that is within the range that *System.decimal* in the .NET client can handle. See *System.decimal.Minvalue* and *System.decimal.Maxvalue*.
- The Java class *URI* is, by default, mapped to the XML schema type *string*.
  If stronger typing is desired and the .NET bindings are to be generated by the *DataContractSerializer*, the field(s) in question can be annotated with @XmlSchemaType(name=”anyURI”), which will cause mapping to the XML schema type anyURI, which is then mapped to the .NET type *System.Uri*.
  Note! *XmlSerializer* will map the Java class *URI* to the XML schema type *string* regardless of any *@XmlSchemaType* annotation.
- The mapping of the Java class *javax.xml.datatype.Duration* is mapped to the XML schema type *duration*. Mapping of *duration* to a .NET type depends on the serializer being used:
  DataContractSerializer maps it to *System.TimeSpan*.
  XmlSerializer maps it to *System.string*.

The .NET utility *System.Xml.XmlConvert* can be used to convert between the *System.string* and the *System.TimeSpan* types.

```
System.string stringTimeSpan = System.Xml.XmlConvert.ToString(new
System.TimeSpan(23, 0,0));
```

- Binary Java types, such as *java.awt.Image*, *javax.xml.transform.Source* and *javax.activation.DataHandler* maps to the XML schema type *base64Binary*, which .NET in turn maps to *byte[]*.
  In Java, the annotation *@XmlMimeType* can annotate field(s) causing the XML schema to add the attribute *expectedContentTypes* to the element in question.
  This information will, however, be ignored by the .NET serializers.
- The Java class *java.xml.datatype.XMLGregorianCalendar* is preferred over *java.util.Date* and *java.util.Calendar*. This since *XMLGregorianCalendar* provides more complete XML support.
  Additionally, fields of the typ *XMLGregorianCalendar* may be annotated with the *@XmlSchemaType* annotation, for instance *@XmlSchemaType(name="dateTime")*, in order to provide stronger typing.
- Use the Leach-Salz variant of UUID at runtime.
  The Java class *java.util.UUID*, which maps to the XML schema type *string*, has constructors that allow for creation of any variant of UUID, the methods in the class are for manipulating the Leach-Salz variant.
- A typed Java variable, for instance *Shape<T>*, will be mapped to the XML schema type *anyType*, which in turn will be mapped to the .NET type *System.Object*.
- By default, collection types maps to an XML schema element with the attributes *minOccurs="0"* , *maxOccurs="unbounded"* and *nillable="true"*.
  .NET will, in turn, map this XML construct to *System.Nullable<T>[]*.
  When marshalling, JAXB will always marshal a null value using the XML schema attribute *xsi:nil=true*.
- If a collection type is annotated with the *@XmlElement(nillable=false)* annotation it will be mapped to an XML schema element with the attributes *minOccurs="0"* , *maxOccurs="unbounded"*. .NET will map such an element to T[], an array of the type in question.
- By annotating a connection with the *@XmlList* annotation, JAXB will map it to a list of XML values (example of a XML list with four numbers: <items>1 2 3 4</items>).
  .NET will map an XML list to a *System.string* type.
- Annotating a field or property using the *@XmlElement* annotation cause it to be mapped to an XML element. This is also the default behaviour in the absence of other JAXB annotations. Using the parameters of the *@XmlElement* annotation, the name of the XML element can be specified, one can indicate if the element is required or not etc.
- Using the *@XmlAttribute* annotation will cause a field or property in a Java class to be mapped to an XML schema attribute. .NET binds an attribute to a property.
- Annotating a field or property with the *@XmlElementRefs* annotation will cause it to be mapped to the XML <choice> element, containing a number of <xs:element ref=.../> elements. .NET will map a <choice> element to a property with the default name *item*.
- Avoid annotating Java classes with the *@XmlType(name="")* annotation that,as in the example, has an empty name. This will cause the Java class to be mapped to an anonymous type, which in turn can cause multiple classes, one for each occurrence of the anonymous type, when mapped to .NET.
  If supplying a name, the result will be mapping to one single .NET class.
- Avoid using *@XmlType(propOrder={})*

Java will map a class annotated with this annotation to an XML complex type containing the <all> element. Due to the restrictions placed on the <all> element, the use of *@XmlType(propOrder={})* is not recommended.

- The *@XmlValue* annotation can be used to cause properties and fields to be mapped to XML <complexType> that contains a <simpleContent> element.
  In .NET, the Java property annotated with the *@XmlValue* annotation will be mapped to a property with the name “value”.
- Fields or properties annotated with the *@XmlAnyElement* annotation will be mapped to the XML schema type <any>, which in turn will be mapped to the .NET type *System.Xml.XmlElement[]*.
- Fields or properties annotated with the *@XmlAnyAttribute* annotation will be mapped to the XML schema type <anyAttribute>, which in turn will be mapped to the .NET type *System.Xml.XmlAttribute[]*.
- Java enum types are mapped to XML schema types constrained by enumeration facets, which in turn maps to the .NET *enum* type.
- The *@XmlSchema* annotation can be used at package level to customize the mapping of a package to an XML schema namespace.
  The following attributes of a XML namespace can be configured using parameters of the annotation. Parameter and attribute names are same, if nothing else stated.
  - elementFormDefault
  - attributeFormDefault
  - targetNamespace (annotation parameter: namespace).
  Namespace prefixes can also be associated with namespaces, as in this example:

```
@javax.xml.bind.annotation.XmlSchema (
  xmlns = {
    @javax.xml.bind.annotation.XmlNs(prefix = "po",
                namespaceURI="http://www.example.com/myPO1"),

    @javax.xml.bind.annotation.XmlNs(prefix="xs",
                namespaceURI="http://www.w3.org/2001/XMLSchema")
  )
)
```

- The *@XmlSchemaType* annotation can be used at package level to customize the mapping of XML schema built-in types.
- The use of the following JAXB annotations is not recommended: *@XmlElementDecl*, *@XmlID*, *@XmlIDREF*.

Java Web Service or Java Web Service Client, WSDL First

There are two possible situations that can occur when developing a Java web service using the WSDL-first approach:

- The WSDL is the result of a contract-first approach.
  This means that it was written by hand and not generated by a Java or .NET tool.
- The WSDL was generated by the .NET *DataContractSerializer*.

Contract First WSDL

When the WSDL is the result of a contract-first approach and we are to develop a Java web service that is to be consumed by WCF client(s), we must make sure that the XML schema features in the WSDL can be consumed by either the *DataContractSerializer* or *XmlSerializer* serialization mechanisms. JAXB does not need this kind of verification, since it supports all XML schema features.
The verification of the WSDL file is done using the svcutil tool, issuing the following command in a terminal window, replacing the name of the WSDL file with the name of your WSDL file:

```
svcutil MyService.wsdl
```

.NET Generated WSDL File

If the WSDL file was generated by the .NET *DataContractSerializer*, and a Java web service or a Java web service client is to be developed, then JAXB customizations should be enabled.
It is recommended by the WSIT Tutorial that both these customizations are applied in a <jaxb:globalBindings> element in an external JAXB configuration file. An example fragment of such a configuration file is shown here:

```
<jaxb:bindings version="2.0"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <jaxb:bindings
        schemaLocation="schema-importedby-wcfsvcwsdl"
        node="/xs:schema">
        <!--
            Set this attribute to false since we have a WSDL
            that was generated by WCF.
        -->
        <jaxb:globalBindings generateElementProperty="false"/>

    </jaxb:bindings>
...
```

The external JAXB configuration file is later supplied to the *wsimport* tool using the -b option.
Please refer to this webpage for more details on customizing JAXB bindings:
http://java.sun.com/webservices/docs/1.5/tutorial/doc/JAXBUsing4.html

The generateElementProperty Attribute

The following XML fragment will, with the *generateElementProperty* attribute set to its default value which is “true”, result in the mapping below:

```
...
    <xs:element name="person" type="tns:Person"/>

    <xs:complexType name="Person">
        <xs:sequence>
            <xs:element name="name" type="xs:string"
                nillable="true" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>
…
```

JAXB generates the *Person* class from the above XML schema fragment:

```
public class Person
{
    JAXBElement<String> getName() {...};
    public void setName(JAXBElement<String> value) {...}
}
```

With the *generateElementProperty* attribute set to “false”, JAX will generate the following version of the *Person* class from the exact same XML schema fragment:

```
public class Person
{
    String getName() {...};
    public void setName(String value) {...}
}
```

Note!
The *generateElementProperty* attribute was introduced in JAXB 2.1.

The mapSimpleTypeDef Attribute

The .NET platform has a number of unsigned datatypes that will map to corresponding unsigned XML schema datatypes:

.NET Datatype	XML Schema Dataype	Maps to Java Datatype
byte	xs:unsignedByte	short
ushort	xs:unsignedShort	int
uint	xs:unsignedInt	long
ulong	xs:unsignedLong	BigInteger

This results in the following two problems:

- The Java datatype can hold values that does not fit in the .NET datatype.
- The serialization by .NET and JAXB of the above mentioned datatypes will not produce a consistent result:
  If, for instance, an *ushort* is serialized by .NET, the XML representation will be an *xs:unsignedShort*. On the Java side, however, the corresponding data is stored in an *int*, which will be serialized to an *xs:int*. Thus the XML representation produced by .NET will be different than that produced by JAXB.

One solution is to use the *mapSimpleTypeDef* attribute in a JAXB configuration file, and set it to "true":

```
<jaxb:bindings version="2.0"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <jaxb:bindings
        schemaLocation="schema-importedby-wcfsvcwsdl"
        node="/xs:schema">
        <!--
            Set this attribute to true since we have unsigned
            types in the WSDL.
        -->
        <jaxb:globalBindings mapSimpleTypeDef="true"/>

    </jaxb:bindings>
...
```

Supplying this JAXB configuration file to *wsimport*, XML schema datatypes like *xs:unsignedShort* will be mapped to their own classes, for instance *UnsignedShort*. Serialization of classes like this will produce results consistent with that of the .NET serialization.
Another way of solving this problem, which is to be preferred, is to avoid datatypes specific to the .NET platform altogether.

WS-I Basic Profile 1.1 Conformance

JAX-WS 2.0 enforces the WS-I Basic Profile 1.1, but there is one situation in which the .NET framework fails to do so:

R2211 in section 4.4.1 of the WS-I Basic Profile 1.1 says:

```
An ENVELOPE described with an rpc-literal binding MUST NOT have
the xsi:nil attribute with a value of "1" or "true" on the part
accessors.
```

In cleartext, this means that, for a RPC/Literal web service, a parameter must not have the null value. .NET clients of a Java web service may fail to honor this rule and pass a null parameter to a RPC/Literal web service method.
Care must be taken when developing .NET web service clients accessing RPC/Literal Java web services to ensure that null parameters are not passed.

11. General Design and Architecture

11.1 Service Oriented Architecture

Describe the characteristics of a service-oriented architecture and how Web services fit this model.

Service oriented architecture is an architectural style that can affect the following areas:

- How software is deployed.
- Design techniques.
- Development methodology.
- Partner/customer/supplier relationships.

SOA Characteristics

Service oriented architecture has the following characteristics:

- Loose coupling.
  SOA applications are constructed from a number of loosely coupled services.
  Loose coupling is achieved by hiding details about services, only providing an interface.
- Reusability.
  Services can be used by multiple applications, both inside one and the same enterprise and crossing enterprise boundaries. New applications can be implemented using existing services. Benefits including: Reducing cost, reducing development time, increased enterprise agility etc.
- Interoperability.
  Services are interoperable, regardless of implementation language, platform etc.
- Scalability.
  Not only the loose coupling between services, but also the fact that they are usually stateless facilitates scalability. Services can also be asynchronous, which enables clients to tend to other matters while awaiting a result from a service.
- Flexibility.
  Loosely coupled, asynchronous, document-based services can be rearranged to meet changing business needs more easily.
- Cost Efficient.
  A standards-based approach to, for instance, integrate legacy systems, applications used by different business partners etc. require less investigation time, development time and effort. SOA enables reuse of business functions exposed as services.
- Dynamic Discoverability.
  A service registry may be used, in order for services to be dynamically discovered.

Web Services in SOA

Web services fit well in the SOA model, according to the above characteristics, because:

- Web services use WSDL documents to specify the interface of a service.
  WSDL is a standardized way of describing a web service interface, which further decouples the interface and the implementation of the web service.
  Benefits: Loose coupling, reusability, flexibility.
- Web services uses standardized, widely accepted, ways of communication that are vendor-independent.
  Benefits: Loose coupling, interoperability.
- A set of standards and recommendations have been specified to ensure interoperability between web service implemented in different languages, on different platforms etc.
  Benefits: Loose coupling, cost efficiency, interoperability.
- Web services can be asynchronous, document based and stateless which results in better scalability.
  Benefits: Scalability, loose coupling.
- Web services is a mature technology with relatively good tool support.
  Benefits: Cost efficiency.
- Web services can be registered in, for instance, a UDDI registry.
  Benefits: Dynamic discoverability.

11.2 Design Patterns and Best Practices

Given a scenario, design a Java EE web service using Web Services Design Patterns (Asynchronous Interaction, JMS Bridge, Web Service Cache, Web Service Broker), and Best Practices.

Asynchronous Interaction

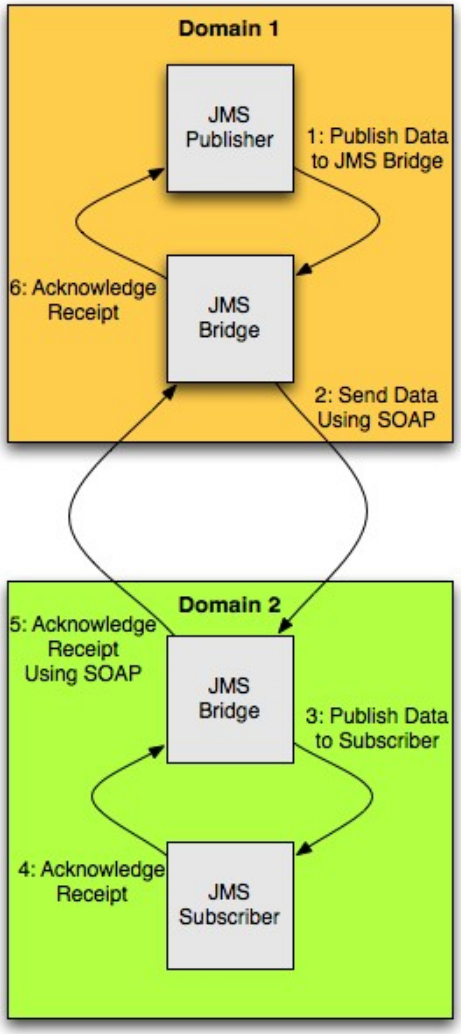
Asynchronous interaction means that clients of a web service issue a request without waiting for a response. If the operation is a request-response operation, the client can either be notified that a response has arrived using a callback or the client can poll for the availability of a response.
The operation may also be a one-way operation, in which case the client will not receive any response.
Asynchronous interaction with a web service may be considered when:

- Response time of the service is expected to be long.
  Synchronous requests with long response times may cause long delays for users of the application issuing the requests.
- Response time of the service may not be predictable.
  Such circumstances may be caused by a process requiring human interaction.
- Users may not be continuously online.
  For instance, when users are connected using a wireless connection.
- More strict requirements concerning reliability and performance exists.

Instead of the client, and thus the user of the client application, being held up waiting for a response, the client can proceed with other matters and, at a later point in time when having received a response, present a result to the user.

JMS Bridge

Messaging systems, such as JMS, provide reliable messaging for business transactions. However, when messages are to cross domains, for instance from one company to another, problems may arise due to the use of different messaging products.
In such a case, web service technology can be used to form a bridge between the two messaging systems. The following picture shows how a message from a JMS publisher is forwarded from one domain to another domain, using a JMS Bridge.



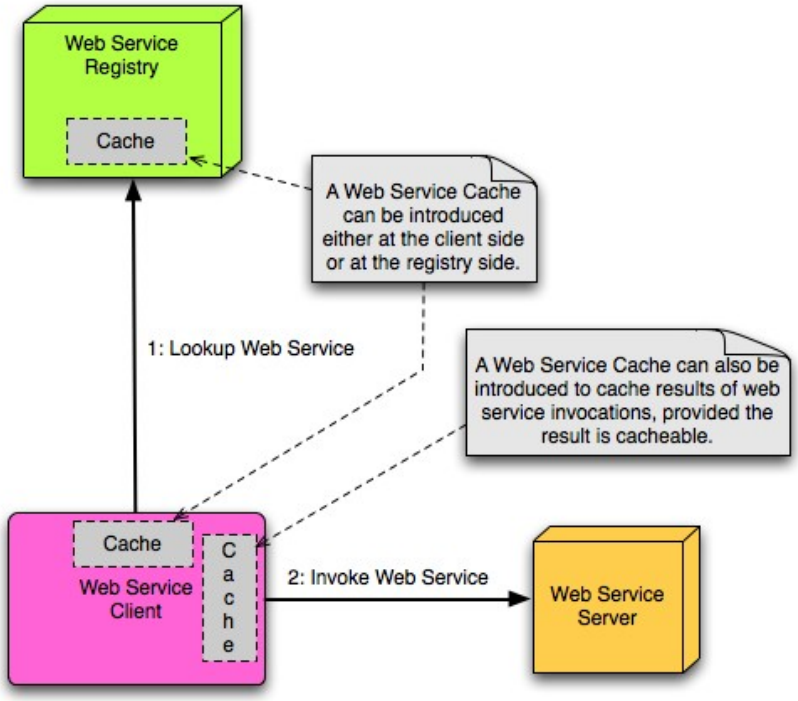
Additionally, the JMS Bridge may implement a mechanism for resending data, in the case that the receiving JMS Bridge component is temporarily unavailable, and a logging mechanism, for tracking purposes.

Drawbacks

The main drawback with the JMS Bridge design pattern is that introduces overhead due to the extra layer introduced by the bridge and the reliable messaging mechanism used by the bridge, which may cause problems in time-critical applications.

Web Service Cache

In an environment where business service and content providers often are added, changed or removed, web service clients often need to look up the web service(s) it uses in a registry.
While this introduces flexibility, it comes at the price of overhead. Such overhead can be minimized by introducing a cache either in the client or in the service registry.
Additionally, some services produce results that may be cached for some period of time, such as foreign exchange rates. Results of such service can be cached in the web service client.



Benefits

- Reduced network traffic.
- Increased performance.

Drawbacks

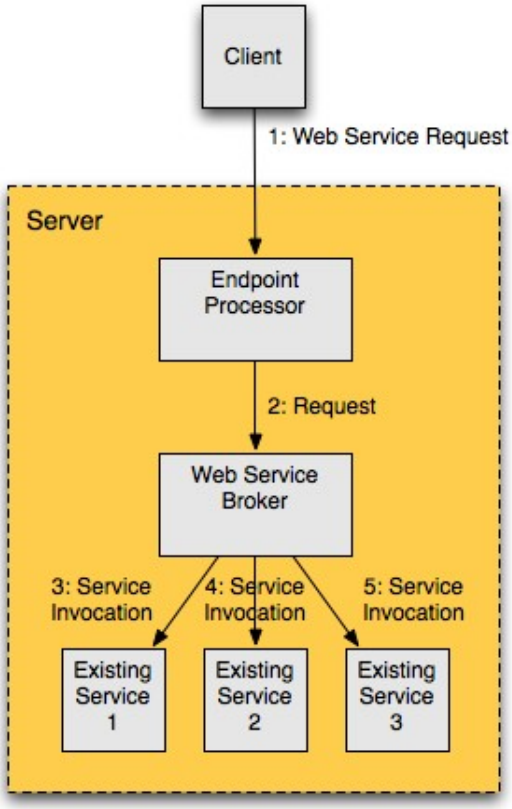
A cache introduces complexity and the following issues need to be considered:

- How to keep data synchronized with data in registry/data produced by web service?
- How to handle stale data in the cache?
- Adds design complexity where cache is introduced.
- How to handle recovery of data in cache in the case of corruption?

Web Service Broker

The Web Service Broker pattern can be used for the following reasons:

- One or more existing services are to be exposed as a web service.
  These services are not necessarily web service, but may be POJOs, EJBs, legacy systems or some other kind of service.
- One or more services need to be exposed in a heterogenous environment.
  For example, a number of services implemented in Java need to be accessed by one or more .NET clients.
- Monitoring and, optionally, usage restrictions are to be imposed on existing services.
  For example, when the system is under heavy load, the Web Service Broker will restrict invocation of the underlying services.
- One or more existing services need to be adapted to business requirements.
  The services may originally lack some capabilities needed to fulfill the new requirements. The Web Service Broker can use existing services and implement additional, required, capabilities.



The above figure shows the flow of a web service request that is handled by a Web Service Broker, which consists of the following parts:

Part	Description
Endpoint Processor	Receives incoming request and, for instance in the case of the request being a SOAP request, processes the SOAP message to extract parameters etc. Endpoint processors are usually part of the runtime system, as is the case with JAX-WS, but may optionally be custom written.
Web Service Broker	Contains the logic handling requests. Coordinates requests to backing services etc. May also contain implementation of additional capabilities missing in backing services, monitoring functionality etc.
Existing Services	This may be: POJOs, EJBs, facades to other systems, legacy systems, web services etc. etc.

Benefits

- Enables reuse of existing services.
- Enables integration in heterogenous environments.
- Monitor access to existing service(s).
- Restrict access to existing service(s).

Drawbacks

- Introduces an additional layer between client and service which may affect performance.
- Protocols such as SOAP over HTTP may affect network performance negatively.
- Existing remote services need to be adapted to provide local access.

Best Practices

Adopting best practices when designing and developing web services has the following advantages:

- Enables building of reusable components.
- Enables building of shareable services.
- Improves developer productivity.
- Reduce development time.
- Reduce infrastructure and support cost.
- Increase quality and reduce risk.
- Increase scalability and availability.

Some best practices, when developing web service, are:

- Follow the WS-I Basic Profile.
  There are tools that verifies compliance with the Basic Profile etc.
- Use WSDL-first development.
  This makes it easier to ensure compatibility in a heterogenous environment and increases stability of the web service interface.
- With non-trivial services, separate service implementations in two layers; an interaction layer and a processing layer.
  The interaction layer consists of:
  - Web service endpoint interfaces and implementations.
  - Logic for preprocessing and delegating requests to processing layer.
  - Logic for formulating responses.
  The processing layer contains:
  - Business logic used to process client requests.
  - Logic that integrates with existing web services or systems.
  Possible advantages of of such a separation are the possibility to introduce caching between the two layers, allowing the processing layer to support different kinds of clients (for instance, a SOAP web service and a RESTful JSON web service).
- Find a balance between flexibility and consolidation of web services to find a good granularity of the services.
  Fine grained services are flexible, but increases overhead and decreases performance.
  Too coarse grained services may become inflexible, from the clients point of view.
- Carefully choose parameter types and return types of service methods.
  Parameters and return values must be mapped to XML or some other representation.
  This will have a significant effect on portability and interoperability of the service.
  Refer, for instance, to section 10.5 on WCF and Java Web Service Interoperability.
- Consider how errors, faults and exceptions are to be handled and what result such conditions will produce.
  Of special interest is how clients running on other platforms, such as .NET, handles errors reported by a web service implemented in Java. Such clients are likely to implement a different mechanism for handling errors.
  For instance, avoid allowing Java exceptions to propagate out of the web service. Instead, wrap such exceptions in a meaningful service-specific exception.

(continued on next page)

- When implementing web service handlers, keep the following in mind:
  - Do not implement business logic or processing related to specific request/responses in a handler.
  - Do not store client-specific state in a handler.
  - Bear in mind that handlers are applied to all the requests and responses of a port and can thus result in a significant performance penalty.
  - Handlers should not alter SOAP messages as to no longer comply with the WS-I Basic Profile.
- Prefer stateless services over stateful.
- Design endpoints to be idempotent.
  Idempotency means that repeated execution of identical requests to a web service have the same effect as issuing a single request. Designing endpoints to be idempotent avoids problems resulting from duplicated messages, which can occur due to clients retrying request, network problems, bugs etc.

11.3 Web Service Interaction Results

Describe how to handle the various types of return values, faults, errors, and exceptions that can occur during a Web service interaction.

References:
JAX-WS 2.1 Specification, sections 2.5, 3.7, 4.2.4, 6.4, 10.2.2, 11.2.1.
https://blueprints.dev.java.net/

In this section, we will examine how different results that can occur during a web service interaction are handled. The results are divided in two groups:

- Return Values
  If the web service interaction resulted in a return value, then it is considered to have completed normally, without errors.
- Faults, Errors and Exceptions
  Errors occurring during a web service interaction results in exceptions, which in turn gives rise to faults.

Return Values

Web services can return either XML data, values or Java objects. The latter two are created, or extracted, from XML data by the JAX-WS runtime.

Java Objects and Values

Comparing web service invocations with invocations of local methods, there is one important difference to keep in mind:

A Java object returned by a web service is always a copy of an object that resides, or resided, on the server side.

The web service runtime reconstructs one or more objects by unmarshaling XML data received in response to a web service request. This results in the following:

- The data on the client side will become stale if the data on the server side is changed.
- Changes to the object(s) on the client side will not be reflected on the server side.
- Depending on the marshaling technology, the platform on which the server is implemented and the platform on which the client is implemented, there may be a difference between the type of the object(s) used to create the representation on the server side and the resulting object(s) on the client side.
  See section 10.5 on WCF and Java Web Service Interoperability above for examples.

Results from web service invocations returning values or Java object can be processed as any other Java method invocation. An example is a controller class in a web application invoking a web service and then making the data available to a JSP, which in turn renders a display of the data.

XML Data

Another use case is if the web service invocation returns XML data. In such a case, an XSL stylesheet can be used to transform the XML data into a HTML document
Tags for handling XML data is also available to JSP pages.

The Java Blueprints gives the following advice, regarding the handling of XML documents:

- In applications having a web tier it is recommended to use JSP to present the data from a web service to the user.
- In applications that does not have a web tier, where JSP is not available, XSLT transformations are recommended.

Faults, Errors and Exceptions

Faults

Faults occurring as a result of a web service invocation will be mapped as follows by JAX-WS:

- If possible, a service-specific exception must be thrown.
- If the fault cannot be mapped to a service-specific exception, then a *ProtocolException* or one of its subclasses must be thrown.

Thus, if not processing raw SOAP messages, faults will be mapped to Java exceptions and the exceptions can be handled as described in the section on exceptions below.
If processing raw SOAP messages, then code like in the following example can be used to determine whether a SOAP message is a fault message or not:

```
SOAPMessage theMessage;

// SOAP message obtained from somewhere.

if (theMessage.getBody().hasFault())
{
    // Process SOAP fault message here.
} else
{
    // Process normal SOAP message here.
}
...
```

Faults can also be processed in handlers. The interface *javax.xml.ws.handler.Handler* contains a method *handleFault* that will be invoked when a fault is passing through the handler.

Errors

A web service should try to avoid throwing system exceptions. As in the section on exceptions, such exceptions will result in a *ProtocolException*, or a subclass of *ProtocolException*, being thrown on the client side. Such exceptions are unchecked and provide the client with little information on the cause of the error.

Exceptions

Two kinds of exceptions can occur as the result of a web service invocation:

- System Exceptions
- Service-Specific Exceptions

JAX-WS defines the following standard, unchecked, exceptions:

Exception	Description
javax.xml.ws.WebServiceException	Runtime exception thrown by JAX-WS APIs when error during local processing occurs.
javax.xml.ws.ProtocolException	Base class for runtime (unchecked) exceptions related to specific protocol bindings.
javax.xml.ws.soap.SOAPFaultException	Subclass of *ProtocolException* used in connection with SOAP binding. An object implementing *SOAPFault* containing the fault from the body of the SOAP message can be obtained using the *SOAPFaultException.getFault()* method.
javax.xml.ws.http.HTTPException	Subclass of *ProtocolException* used in connection with HTTP binding. The HTTP status code can be obtained using the *HTTPException.getStatusCode()* method.

All methods in a service endpoint interface, or service endpoint implementation class, can throw *javax.xml.ws.WebServiceException* and any number of service-specific exceptions.

System Exceptions

System exceptions are all unchecked Java exceptions plus *java.rmi.RemoteException* and all subclasses of *RemoteException*. Possible causes for a system exception being thrown as the result of a web service invocation are:

- The invocation fails due to, for instance, the URI of the service being bad.
- Service not being accessible.
- Network communication problems.
- Errors beyond the control of the web service application.

System exceptions occurring on the server side during a web service invocation will result in an appropriate subclass of *ProtocolException* being thrown.

Example:
If a system exception is thrown during the execution of a SOAP web service, a SOAP fault will be generated and, when reaching the client, a *SOAPFaultException* will be thrown.

Tactics for dealing with system exceptions in a web service client may include:

- Retry the web service invocation.
  Retrying the invocation may include a delay before the next invocation is attempted, a limited number of retries etc.
- Redirect the request to an alternative service.
- Notify the user of the client application that an error has occurred.
- Logging the error.
- Translate checked system exceptions to an unchecked Java exception and let the client application handle the unchecked exception.

Service-Specific Exceptions

Service-specific exceptions are all checked exceptions, with the exception of *java.rmi.RemoteException* and all subclasses of *RemoteException*.
Service-specific exception thrown by a web service method are to be mapped to <wsdl:fault> elements declared in the abstract interface of the web service (example). A mapped Java exception class must be annotated with the *@WebFault* annotation.
Possible reasons for service-specific exceptions are:

- Data presented to the service is out of bounds.
  For example, trying to retrieve a customer using a customer id that does not exist.
- Data presented to the service duplicates existing data.
  For example, trying to add a new order with an order number that already exists in the system.
- Data presented to the service is incomplete.

The following are possible ways of handling service-specific exceptions in client applications:

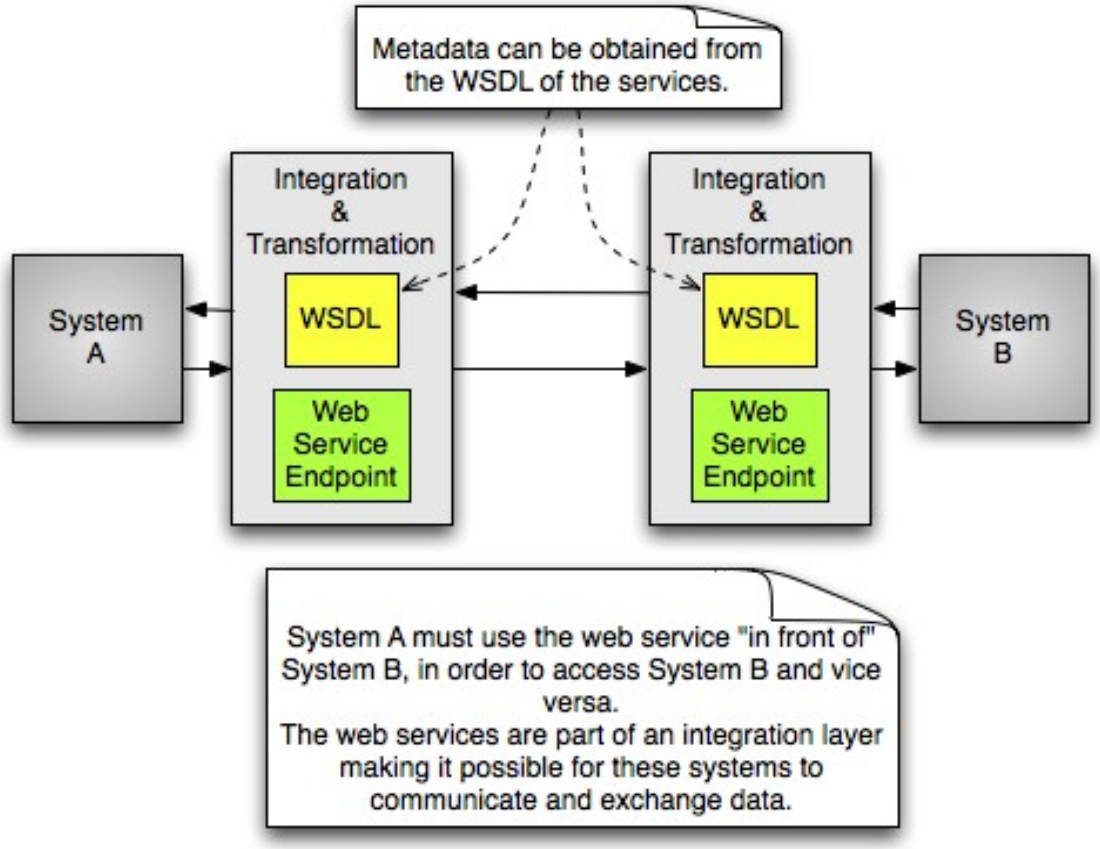
- Wrap the checked service-specific exceptions and throw an unchecked exception.
- Notify the user of the client application that an error has occurred.
- Do not allow service-specific exceptions propagate throughout the client application, but rather wrap them in exceptions specific to the client application.
  This limits the impact of changes in the web service interface.
- Handle exceptions at one single place in the client application.
  Prevents duplication of code and limits impact of changes in the web service interface.

It is recommended to do boundary checking of parameters of web service requests prior to sending the request, in order to minimize the number of web service calls.

11.4 Web Services and Data Integration

Describe the role that Web services play when integrating data, application functions, or business processes in a Java EE application.

References:
Designing Web Services with the J2EE 1.4 Platform



The above figure shows an example how web services can be used to:

- Integrate the data.
  Allowing System A and System B to exchange data, even though the data formats of the systems are different.
- Integrating application functions.
  System A can use functionality provided by System B and vice versa.
- Integrate business processes.
  The integration between System A and System B may be a part of a larger inter-enterprise integration done in order to be able to use and re-use services when providing support for business processes.

Integrating Data

Data integration means that data located in a system is adapted so as to allow access from other systems. The format of the data is likely to vary between the systems, additionally the data models may also be different.

Web Service as an Integration and Transformation Layer

Web service can play the role of an integration and transformation layer, integrating of data from different enterprises and different systems etc.
Web services enables making various kinds of information from different sources available in a manner (XML or JSON) that can be read and understood by clients on different platforms.
A canonical data model, a data model that does not depend on any particular application, can be established, for instance by using XML. New applications are developed to use the canonical data model and legacy systems use transformations, which are part of the integration layer, to adapt their data to the canonical data model when communicating with other systems.
Using XML to establish the canonical data model has the following advantages:

- XML schemas can be enforced.
  XML documents can be validated using an XML schema to ensure that they correspond to the canonical data model.
- XML documents can be transformed and converted to other formats.
- XML is both platform and programming-language independent.

Web Services as a Metadata Provider

Web services can play the role of a metadata provider when integrating data.
Metadata is data about the format of data. Such metadata is present in a WSDL file, which not only contain information about functionality supplied by the service, but also the data formats used when communicating with the service.
Additionally, WS-Policy can be used to further declare the policies of a service regarding security, quality of service etc.

Integrating Application Functions

Integration of application functions means integrating new applications with existing or legacy applications.

Web Services and User Experience

Web services can aid in integrating application functions to provide a more seamless experience of multiple applications for users.
Example: Company A is bought by company B. Company B can use web services to provide single sign on, allowing customers from the both companies to use the online services of the other company after having logged in to their original account.

Web Service as a Reuse Facilitator

Web services can integrate application functions in an enterprise, and across enterprise boundaries, to avoid the same functionality being implemented in multiple locations.
This increases reuse, reduces cost and increases flexibility for the enterprise.

Web Services as an Integration Layer

Web services can play the role of an integration layer used to integrate new application functions with existing application functions. This layer acts as a facade behind which details of the underlying (legacy) system is hidden. The integration layer also establishes clearly defined boundaries between the different systems.
Finally, web services provide a opportunity to control and monitor access to the underlying systems.

Integrating Business Processes

Integration of business processes involves integration of the existing systems of an enterprise in order to support a set of business processes. A business process is a number of steps that together form a business task or a business function.
For instance, the business process booking a hotel room may include the step credit card validation.
Integration is often accomplished by the exchange of XML documents between business processes according to certain business rules.

Web Services as Enterprise Service Providers

Web service, in combination with a web service registry, can provide a dynamic way to maintain a catalog of services provided by business partners or services an enterprise supplies to business partners.
For instance, a credit card company makes credit card payment services available to a hotel booking company and an online shopping company.
Functionality is provided by exposing interfaces in a standardized way that is platform and technology independent.

Web Services as a Communication Facilitator

Web services, being platform agnostic, can provide a way to tie together, for instance, a hotel room booking service with a credit card payment service, enabling customers to use their credit card to pay for hotel rooms. The hotel room booking company and the credit card company may be two different companies with systems running on different platforms.

Gains

- Web services can provide a more cost effective way to integrate business partners of varying sizes.
  For instance, by using existing communication infrastructure.
  Web services does not require a big, up-front, investment in a surrounding infrastructure.
- Web services can be used on a variety of platforms.
  Being standardized, web services can be used on many different platforms, thus is less likely to require time or money to be invested in new hardware or software.

Drawbacks

There are some drawbacks of using web service technology to integrate data, application functionality and business processes:

- Introduction of an additional layer causes delays due to, for instance, validation and processing of XML data.
- Web services is a rather complex technology, which may require investments in, for instance, education.

12. Endpoint Design and Architecture

12.1 Procedure or Document Style

Given a scenario, design Web Service applications using information models that are either procedure-style or document-style.

Resources:
http://java.sun.com/developer/technicalArticles/xml/jaxrpcpatterns/
Designing Web Services with the J2EE 1.4 Platform , section 8.3.4

Below, some motivating factors for choosing either document or procedure-style web services are presented. These are not absolute rules, but guidelines.

Procedure-style web services may be preferred when:

- The service is to be used within one and the same enterprise.
- To avoid the complexity of creating and manipulating XML documents.
  Requests and responses will still contain XML data, but it will be transformed to Java objects by, for instance, JAXB. The client and server applications thus do not need to be aware of the representation (XML) use to pass the Java objects over the wire.
- The exchanged data is less complex.
- The payload of a (SOAP) request need not to be validated in its entirety.
- A synchronous transport protocol, like HTTP, is used.
- The service is able to produce a results in a shorter time.
- Memory and/or CPU resources are scarce.
  Parsing an XML document and maintaining the model in memory requires more memory compared to procedure-style parameters.

Document-style web service may be preferred when:

- The service is to be offered to other, external, enterprises.
  This is a so-called business-to-business scenario.
- A higher level of stability in the service interface is desired.
- External factors require the exchange of documents that can be verified.
  One example of such an external factor is legal factors.
  Exchanging XML documents enables validation of such documents against well-defined XML schemas.
- The exchanged data is more complex.
- The service need to be able to handle many different types of documents.
- An asynchronous transport protocol is used.
- The service require a longer time before being able to produce a result.
- The entire payload of a (SOAP) request must be validated.
  With procedure-style (RPC) web services, elements wrapping the RPC data does not appear in any XML schema. This prevents validation of the payload to a procedure-style service.

12.2 Service Interaction and Processing Layers

Describe the function of the service interaction and processing layers in a Web service.

References:
Designing Web Services with the J2EE 1.4 Platform , sections 3.4, 3.5, 8.3.5, 8.3.5

When designing a web service, it is generally a good idea to separate it into two layers, a service interaction layer and a processing layer. Some reasons for this kind of separation are:

- Separating responsibilities.
- Concentrate request pre- and post-processing in one single location.
- Avoid exposing (internal) data structures used by the business layer.
  The reason for this is to reduce coupling to the business layer.

The two layers take on different roles as follows.

Service Interaction Layer

Some responsibilities of the service interaction layer are:

- Present an interface to clients of the service.
- Receive requests to the web service from clients.
- Error handling and reporting.
  This include errors occurring as a result of bad in-data, malformed requests etc but also errors occurring in the processing layer.
- Pre-process data received in the request.
  For instance:
  - Validating security credentials.
  This may include invoking security services, such as authentication and authorization.
  - Validating of incoming data.
  For services receiving XML documents, such documents are validated using the appropriate XML schema.
  - Adapting request data to data model used by processing layer.
  For services receiving and processing XML documents, transformation of the received XML data may be necessary.
  For services processing data in objects, transformation of XML data to an object representation is necessary.
- Delegating requests to appropriate logic in the processing layer.
- Post-process request result.
  For instance:
  - Adapting response data from data model used by processing layer to data model used by service.
  - Adapting errors to service-specific faults.
- Generate and send responses to clients.
- Isolating the processing layer from the technology used to receive requests.
  There may be additional ways of invoking the services in the processing layer, for instance using a remote EJB. This enables different kinds of client to use the same processing layer.
- Logging and auditing of requests to the service.
- If desired, implement caching of responses received from the processing layer, in order to avoid unnecessary invocations of the processing layer.

Note that the processing performed in the service interaction layer is processing performed in addition to the processing performed by, for instance, the JAX-WS runtime and JAXB when a request is received.
In the case that the service do not require any preprocessing, the both layers are often merged to avoid complicating the design.

Processing Layer

Responsibilities of the processing layer include:

- Act as a Web Service Broker orchestrating access to one or more underlying services.
- Integrate with enterprise information systems and other web services.
- Process client requests.
  The processing layer contains all the business logic.
- Generate data to be sent with the response to the client, if a response is to be given.

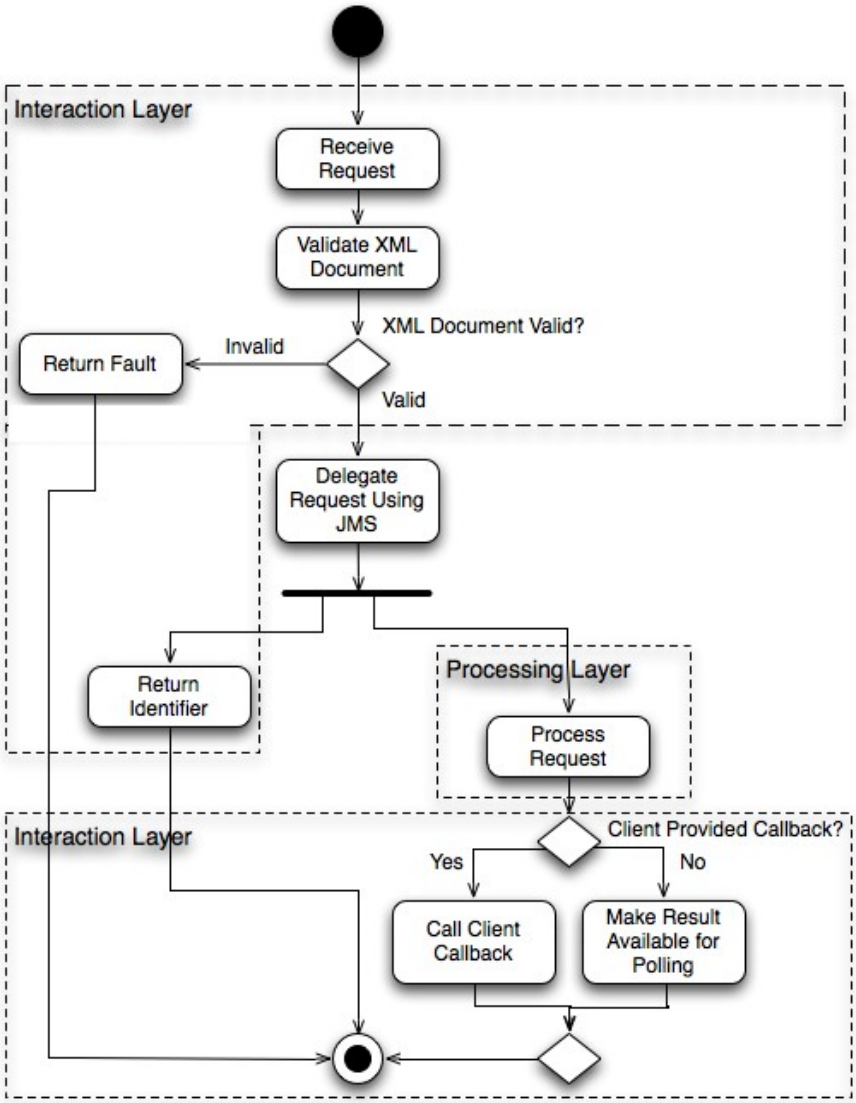
12.3 Synchronous vs Asynchronous

Design a Web service for an asynchronous, document-style process and describe how to refactor a Web Service from a synchronous to an asynchronous model.

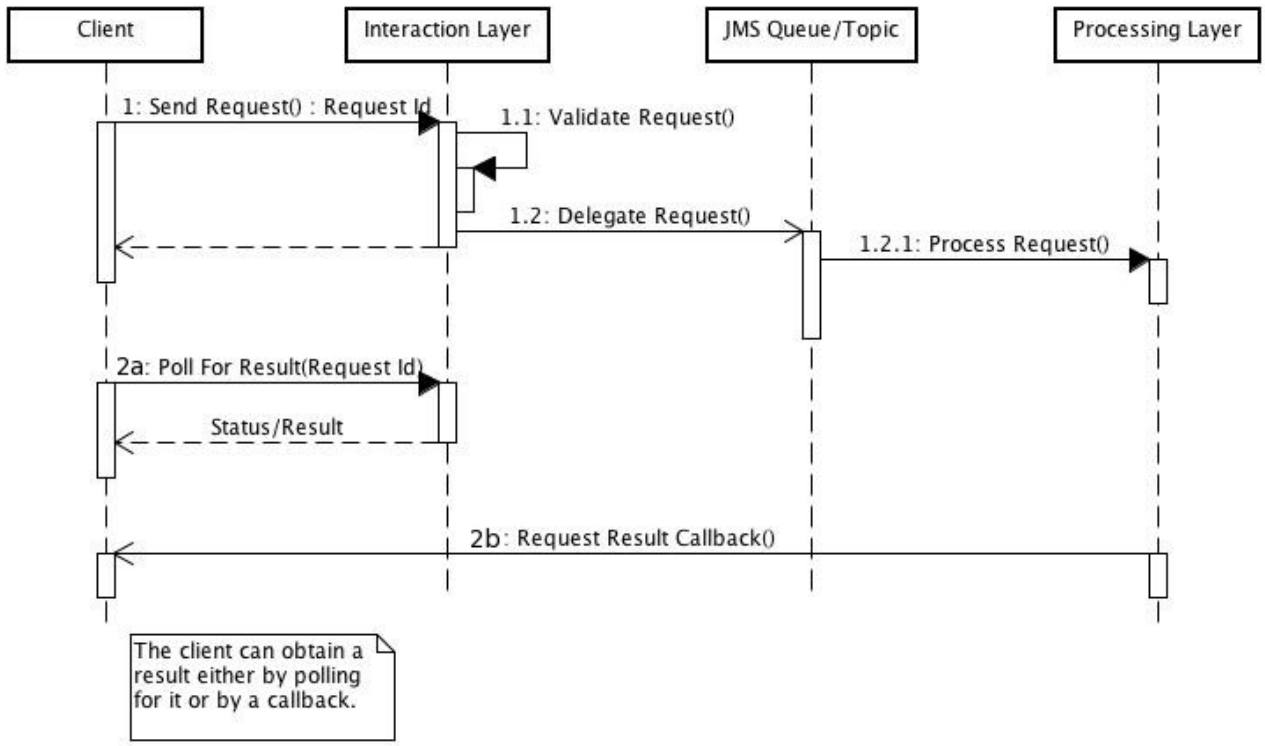
Design an Asynchronous Document-Style Web Service

References:
Designing Web Services with the J2EE 1.4 Platform , section 3.4.3, 8.4, 8.6.3
JAX-WS 2.1 Specification, sections 2.3.4, 8.7.3

In this section, we will look at the design of an asynchronous, document-style, web service. Compare this to asynchronous invocation of a web service described in section 4.8 which invokes a web service that synchronously processes each request and returns a response.
The figure below shows how a request is processed by an asynchronous web service. The identifier return by the service is an identifier that the client later can use when polling for a result of the request, or if a callback mechanism is used, determine which request the response received corresponds to.



Another way of showing the same thing; the figure below shows the client sending a request, the processing of the request and finally obtaining the response. As before, the response can be obtained either by polling or by using a callback mechanism. Both alternatives are show in the figure.



Obtaining a Result Using Polling

Polling of the result means that the client periodically invokes a special operation in the web service, supplying the identifier received as a result of the operation that submitted the XML document. If the web service has finished processing the document, it will return it to the client, otherwise some reply saying that no result is available will be returned.

Obtaining a Result Using Callback

Callback is an alternative to polling for a result, but requires that the client submitting the XML document also is a web service server. When a result is available, the web service to which the XML document was submitted can invoke the client's web service. This has the advantage of the client not having to poll for the result, but has the disadvantage of increasing the complexity of the client.

Handling Errors

Errors occurring during the invocation of an asynchronous web service can be categorized in the the following two categories:

- Errors occurring in the interaction layer.
  Such errors occur prior to a result of the invocation having been reported to the client and can thus be reported as described in section 11.3 above, using a service-specific exception or a fault.
- Errors occurring in the processing layer.
  This kind of errors can further be categorized as being temporary errors or permanent errors.

Errors in the Processing Layer

The different categories of errors in the processing layer can be resolved in different manners.

Temporary Errors

Temporary errors may be caused by temporary network problems occurring when trying to contact a database or another service that the processing layer depends on etc. The following actions can be taken to resolve errors like these:

- Implement a retry mechanism.
  Retry, for instance, connecting to another service a certain number of times, perhaps also incorporating a delay between each retry.
- Implement a failover mechanism.
  If the success of each request is important, a failover mechanism that, if one service fails, will select a backup service to take its place, can be implemented.
- Notify the client that a temporary error has occurred.
  The client can then decide whether or not to retry the request.

Permanent Errors

Permanent errors may be caused by the lack of proper credentials to use some resource, invalid data in the request etc. When errors like these occurs, they usually require intervention from a human; a system administrator, the person responsible for supplying request data etc. Actions taken depends on the nature of the error:

- Failure of underlying resource(s).
  In the case of permanent failure to use some resource the service depends on, an administrator should be notified, for instance, using email.
  The client of the service also need to be informed and the user of the client told that the request failed and have to be resubmitted if the user wants it to be processed.
- Invalid data in the request.
  In the case of invalid data in the request, the client needs to be informed that further processing of the request is impossible due to invalid data and, if possible, point out the offending data. The client can in turn inform the user, which can correct the invalid data before the request is resubmitted.

Notifying the Client

Due to the request having been left the interaction layer and having been submitted for asynchronous processing in the processing layer, the client is not waiting for a result. Informing the client about errors may be done in the same manner as when delivering the result of the request, that is either when the client polls for a response or using a callback mechanism.

Refactor Synchronous to Asynchronous Web Service

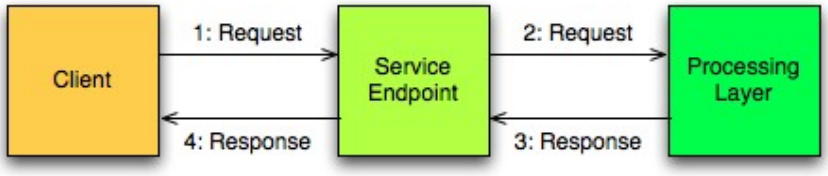
References:
Designing Web Services with the J2EE 1.4 Platform , section 8.4.3

Motivation

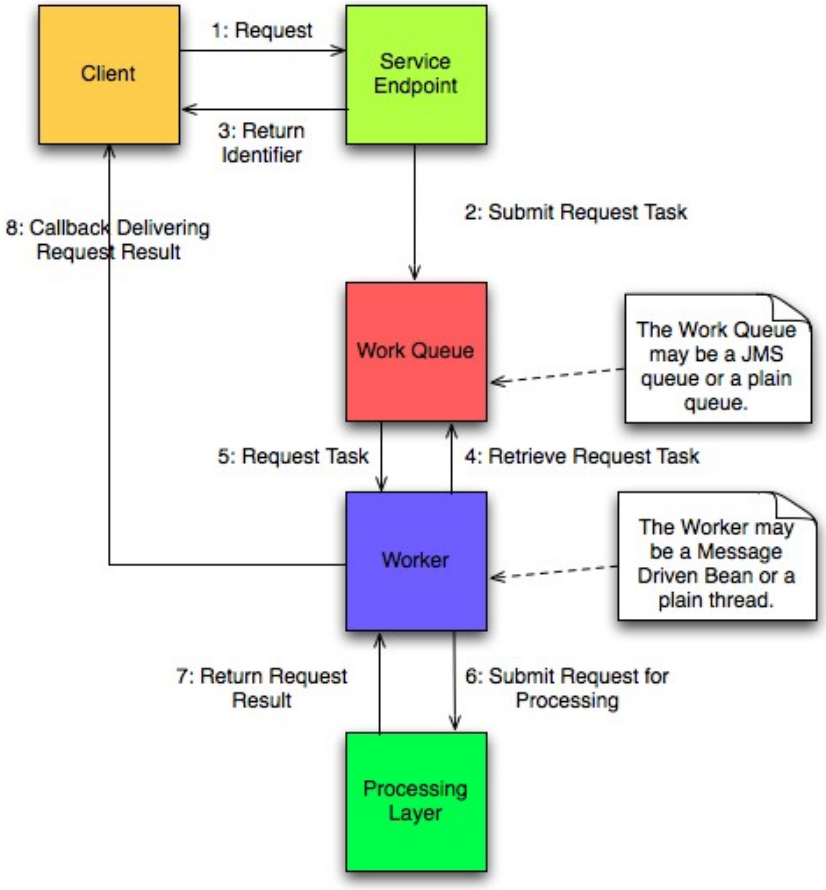
The main motivation for refactoring a web service from being synchronous to being asynchronous is scalability; an asynchronous web service retain its responsiveness when under higher load, though it will not process the requests faster. Requests can be queued in the service, while the client is able to proceed with other matters, and processed by the service when being able to, thus avoiding overload.

Before and After

The following figures shows how requests to a web service are handled before and after refactoring to an asynchronous web service:



A synchronous web service processing a request from a client.



An asynchronous web service processing a request from a client.

Note that the client can also, as described in the section above on designing asynchronous web services, obtain the result by polling. When callback is used to deliver the response to clients, returning an identifier as a result of the request is optional.

Refactoring Step by Step

Refactoring a synchronous service to become asynchronous may be done using the following procedure:

- Change the return type of the web service operation to be refactored.
  If a callback mechanism is to be used, then void can be used.
  If a polling mechanism is to be used or if the client needs to know about the identifier of the request, then the return type is to be the identifier type.
- If a callback mechanism is to be used, then add a parameter to the web service operation to be refactored that specifies the endpoint to deliver the response to.
- Move the code in the service endpoint operation to be refactored that delegates the request to the processing layer to the worker.
  Retain any code that validates the request in the endpoint.
- Implement code in the operation to be refactored in the service endpoint to put the request in a work queue and, optionally, associate an identifier with the request.
- Implement code in the worker that either calls back to the client and delivers the response or makes the response available for polling.

Note that clients of the refactored service also needs to be refactored using the following procedure:

- Change the invocation of the refactored web service so that it, instead of a response, accepts a void return value or a request identifier.
- If a callback mechanism is to be used, add a parameter to the invocation of the refactored web service operation specifying where to deliver the response to.
- If a callback mechanism is to be used, implement it.
- If a polling mechanism is to be used, implement it.

There are a few choices regarding how to implement the work queue and the worker:

- Using the JavaSE *Queue*, *ThreadPoolExecutor* and *Callable* or *Runnable*.
- Using JMS with a message-driven EJB.

12.4 Web Service Client Impact

Describe how the characteristics, such as resource utilization, conversational capabilities, and operational modes, of the various types of Web service clients impact the design of a Web service or determine the type of client that might interact with a particular service.

References:
Designing Web Services with the J2EE 1.4 Platform , section 5.2, 5.4

Resource Utilization

Different kinds of web service clients have require different amounts of resources and different kind of platforms on which web service clients run can provide different amounts of resources. For instance, a client running in a cellphone differs significantly comparing to a client running in a JavaEE container. The availability of resources may limit the amount of data a client is able to process and the amount of state data that can be stored.
Limitations in the clients will impact the design of the service, for instance it may have to provide more fine-grained services to clients with less processing capabilities, it may have to retain conversational state on the server side etc.

Conversational Capabilities

The ability of a web service and its clients to maintain a conversation-like interaction requires storing information related to the conversation. Maintaining a conversational state, regardless of whether it is done at the client or server side, makes the coupling between client and server tighter. Some of the issues that need to be considered in connection to this are:

- Does the conversational state need to be persisted?
  If so, then the following questions need to be considered:
  - How much storage space is needed?
  - How to handle expiration of conversational state data?
- Scalability of a server maintaining state.
  Maintaining a conversational state on the server side will affect its scalability.
- Token passed between client and server to identify a certain conversational state.
- Client developers need to know what the conversational state contains.
- When does a conversational state expire?
- How to manage state shared between multiple clients?
  For instance, if a client A retrieves an order from a web service in order to process it and, before client A has submitted the processed order to the service, client B requests the same order. Client A and B will have identical copies of the order and, when both have submitted the order to the web service, the changes performed by one of the clients will be overwritten. A solution is to lock the data in the service as it is retrieved by one client and maintain the lock until the client in question submits the processed data.

Operational Modes

Different kinds of clients operate in different kinds of modes, concerning accessing web services. Examples are:

- Continuous access.
  The web service client more or less continuously issues requests to one or more web services.
- Intermittent access.
  The web service client occasionally issues request, for instance in response to user actions.
- Batch access.
  The web service clients accumulates a number of requests that, at some point in time, are sent in a batch to the web service.

Reasons for using different operational modes may be that some clients operate under poor, or very restricted, network conditions or that some clients only require to issue occasional requests.

Web Service Client Types

There are basically four different types of web service clients:

- Java EE Clients
  An EJB, another web service or a POJO in a JavaEE application.
- Java SE Clients
  A standalone application.
- Java ME Clients
  A Java application running on a mobile device.
- Non-Java Clients
  For instance, .NET client.

Java EE Clients

Java EE clients run inside a Java EE container, which provides a number of services. The following are some characteristics of Java EE web service clients:

- Container provides declarative security.
- Container manages transactions.
- Container provides lifecycle management (EJB endpoints).
- Container provides thread management (EJB endpoints).
- Container manages creation and injection of web service proxies.
- Client-side web service deployment information can be configured using deployment descriptors, which enables reconfiguration without changing code.
- Container provides additional services, such as asynchronous messaging (message driven EJBs).

Choosing a Java EE client can affect the design of the web service and puts some limitations on the client as follows:

- If multiple web service clients run in the same container, then this may affect the size of the conversational state each client is able to maintain.
  The web service must be designed as to be scalable, despite a larger number of clients.
- With multiple clients in the same container, there will be limitations to the amount of data each client is able to retrieve from the web service.
  Thus the web service may have to impose restrictions on the amount of data retrieved.
- Java EE clients are more likely to require continuous access to web services.
  Web services must be designed as to have high availability.
- If the web service client is an EJB, then all the processing of data (retrieved from a web service) must be done in one and the same thread, since thread creation in an EJB is not allowed.
  Java EE clients are less likely to be able to process very large amounts of data in an efficient manner.

Java SE Clients

Java SE clients run as standalone applications in a rather limited environment. The following are some characteristics of Java SE web service clients:

- Have to create and maintain proxies that accesses the web service.
- Have to implement things like transaction management, thread management, lifecycle management etc. themselves, as there is no container providing these services.
- Typically have more resources, such as processing power and memory, at their disposal.
- Can provide a rich user interface typically more suitable for viewing and processing of larger sets of data.
- Requires only intermittent access to web services.

Choosing a Java SE client can affect the design of the web service in the following ways:

- Web service clients may maintain a larger amount of state data that may be enclosed with every request to the web service.
- Care must be taken as to ensure data consistency.
- If clients can operate in either connected or disconnected mode, this must be kept in mind when designing the web service in order to avoid issues related to maintenance of state and/or service synchronization.

Java ME Clients

Java ME clients run in handheld or embedded devices, which puts further limitations on the environment. Such web service clients are chosen mainly for the following reasons:

- Mobility.
- Remote access.
- When users, regardless of location, need to be able to access the web service with very short notice.

The following are some characteristics of Java ME web service clients:

- Less resources, such as processing power, memory and bandwidth, at their disposal.
  Service providers may also charge for network usage depending on amount of data transferred. This also motivates limiting bandwidth usage.
- Connectivity may be sporadic.
  Thus the probability that the delivery of a request or response may fail is higher and this must be taken into consideration.
  The client may be designed to allow for certain operations to be performed offline and later sent in a batch to the web service.
  Client state and synchronization with the service also need to be kept in mind when designing the web service.
- Network latency may be high.

The following are some design issues related to a web service with Java ME clients:

- Bandwidth usage must be limited.
  This may result in having to maintain client state in the server, which in turn may affect scalability of the web service.
  This may result in the operations provided by the web service having to be more fine-grained.
- Clients may send requests in batches.
- Clients have very limited processing capabilities.

Non-Java Clients

The main issue with non-Java web service clients is to maintain interoperability between the clients and the web service.
Additionally, depending on the client kind, there will be issues similar to those described in the above sections on Java EE, SE and ME clients.