CHAPTER 1: Developing a Consumer with Celtix

Table of Contents

Generating the Stub Code

The starting point for developing a service consumer (or client) in Celtix is a WSDL contract, complete with port type, binding, and service definitions. You can then use the `wsdl2java` utility to generate the Java stub code from the WSDL contract. The stub code provides the supporting code that is required to invoke operations on the remote service.

For Celtix clients, the `wsdl2java` utility can generate the following kinds of code:

- Stub code — supporting files for implementing a Celtix client.
- Client starting point code — sample client code that connects to the remote service and invokes every operation on the remote service.
- Ant build file — a `build.xml` file intended for use with the `ant` build utility. It has targets for building and for running the sample client application.

Basic HelloWorld WSDL contract

Listing 1 shows the HelloWorld WSDL contract. This contract defines a single port type, `Greeter`, with a SOAP binding, `Greeter_SOAPBinding`, and a service, `SOAPService`, which has a single port, `SoapPort`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld"
targetNamespace="http://objectweb.org/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://objectweb.org/hello_world_soap_http"
    xmlns:x1="http://objectweb.org/hello_world_soap_http/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <schema targetNamespace="http://objectweb.org/hello_world_soap_http/types"
            xmlns="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
            <element name="sayHi">
                <complexType/>
            </element>
            <element name="sayHiResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMe">
                <complexType>
                    <sequence>
                        <element name="requestType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeOneWay">
                <complexType>
                    <sequence>
```

```
                    <element name="requestType" type="string"/>
                </sequence>
            </complexType>
        </element>
        <element name="pingMe">
            <complexType/>
        </element>
        <element name="pingMeResponse">
            <complexType/>
        </element>
        <element name="faultDetail">
            <complexType>
                <sequence>
                    <element name="minor" type="short"/>
                    <element name="major" type="short"/>
                </sequence>
            </complexType>
        </element>
    </schema>
</wsdl:types>

<wsdl:message name="sayHiRequest">
    <wsdl:part element="x1:sayHi" name="in"/>
</wsdl:message>
<wsdl:message name="sayHiResponse">
    <wsdl:part element="x1:sayHiResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeRequest">
    <wsdl:part element="x1:greetMe" name="in"/>
</wsdl:message>
<wsdl:message name="greetMeResponse">
    <wsdl:part element="x1:greetMeResponse" name="out"/>
</wsdl:message>
<wsdl:message name="greetMeOneWayRequest">
    <wsdl:part element="x1:greetMeOneWay" name="in"/>
</wsdl:message>
<wsdl:message name="pingMeRequest">
    <wsdl:part name="in" element="x1:pingMe"/>
</wsdl:message>
<wsdl:message name="pingMeResponse">
    <wsdl:part name="out" element="x1:pingMeResponse"/>
</wsdl:message>
<wsdl:message name="pingMeFault">
    <wsdl:part name="faultDetail" element="x1:faultDetail"/>
</wsdl:message>

<wsdl:portType name="Greeter">
    <wsdl:operation name="sayHi">
        <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
        <wsdl:output message="tns:sayHiResponse" name="sayHiResponse"/>
    </wsdl:operation>

    <wsdl:operation name="greetMe">
        <wsdl:input message="tns:greetMeRequest" name="greetMeRequest"/>
        <wsdl:output message="tns:greetMeResponse" name="greetMeResponse"/>
    </wsdl:operation>

    <wsdl:operation name="greetMeOneWay">
        <wsdl:input message="tns:greetMeOneWayRequest" name="greetMeOneWayRequest"/>
    </wsdl:operation>

    <wsdl:operation name="pingMe">
        <wsdl:input name="pingMeRequest" message="tns:pingMeRequest"/>
        <wsdl:output name="pingMeResponse" message="tns:pingMeResponse"/>
        <wsdl:fault name="pingMeFault" message="tns:pingMeFault"/>
    </wsdl:operation>
```

```
    </wsdl:portType>

    <wsdl:binding name="Greeter_SOAPBinding" type="tns:Greeter">
        ...
    </wsdl:binding>
    <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
            <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Listing 1: HelloWorld WSDL Contract

The `Greeter` port type from Listing 1 defines the following WSDL operations:

- `sayHi` — has a single output parameter, of `xsd:string` type.
- `greetMe` — has an input parameter, of `xsd:string` type, and an output parameter, of `xsd:string` type.
- `greetMeOneWay` — has a single input parameter, of `xsd:string` type. Because this operation has no output parameters, Celtix can optimize this call to be a oneway invocation (that is, the client does not wait for a response from the server).
- `pingMe` — has no input parameters and no output parameters, but it can raise a fault exception.

Listing 1 also defines a binding, `Greeter_SOAPBinding`, for the SOAP protocol. In practice, the binding is normally generated automatically — for example, by running either of the Celtix `wsdl2soap` or `wsdl2xml` utilities. Likewise, the `SOAPService` service can be generated automatically by running the Celtix `wsdl2service` utility.

Generating the Stub Code

After defining the WSDL contract, you can generate client code using the Celtix `wsdl2java` utility. Enter the following command at a command-line prompt:

```
wsdl2java -ant -client -d ClientDir hello_world.wsdl
```

Where *`ClientDir`* is the location of a directory where you would like to put the generated files and `hello_world.wsdl` is a file containing the contract shown in Listing 1. The `-ant` option generates an ant `build.xml` file, for use with the `ant` build utility. The `-client` option generates starting point code for a client `main()` function.

The preceding `wsdl2java` command generates the following Java packages:

- `org.objectweb.hello_world_soap_http`

  This package name is generated from the `http://objectweb.org/hello_world_soap_http` target namespace. All of the WSDL entities defined in this target namespace (for example, the `Greeter` port type and the `SOAPService` service) map to Java classes in the corresponding Java package.

- `org.objectweb.hello_world_soap_http.types`

  This package name is generated from the `http://objectweb.org/hello_world_soap_http/types` target namespace. All of the XML types defined in this target namespace (that is, everything defined in the `wsdl:types` element of the HelloWorld contract) map to Java classes in the corresponding Java package.

The stub files generated by the `wsdl2java` command fall into the following categories:

- Classes representing WSDL entities (in the `org.objectweb.hello_world_soap_http` package) — the following classes are generated to represent WSDL entities:
  - `Greeter` is a Java interface that represents the `Greeter` WSDL port type. In JAX-WS terminology, this

Java interface is a *service endpoint interface*.

  - `SOAPService` is a Java class that represents the `SOAPService` WSDL service element.
  - `PingMeFault` is a Java exception class (extending `java.lang.Exception`) that represents the `pingMeFault` WSDL fault element.

- Classes representing XML types (in the `org.objectweb.hello_world_soap_http.types` package) — in the HelloWorld example, the only generated types are the various wrappers for the request and reply messages. Some of these data types are useful for the asynchronous invocation model.

Implementing a Celtix Client

This section describes how to write the code for a simple Java client, based on the WSDL contract from Listing 1. To implement the client, you need to use the following stub classes:

- Service class (that is, `SOAPService`).
- Service endpoint interface (that is, `Greeter`).

Generated Service Class

Listing 2 shows the typical outline a generated service class, *`ServiceName`*, which extends the `javax.xml.ws.Service` base class.

```
// Java
public class ServiceName extends javax.xml.ws.Service {

    ...
    public ServiceName(URL wsdlLocation, QName serviceName) { }

    public ServiceName() { }

    public Greeter getPortName() { }
    .
    .
    .
}
```

Listing 2: Outline of a Generated Service Class

The *`ServiceName`* class in Listing 2 defines the following methods:

- Constructor methods — the following forms of constructor are defined:
  - *`ServiceName`*`(URL wsdlLocation, QName serviceName)` constructs a service object based on the data in the `serviceName` service in the WSDL contract that is obtainable from `wsdlLocation`.
  - *`ServiceName`*`()` is the default constructor, which constructs a service object based on the service name and WSDL contract that were provided at the time the stub code was generated (for example, when running the Celtix `wsdl2java` command). Using this constructor presupposes that the WSDL contract remains available at its original location.
- `get`*`PortName`*`()` methods — for every *`PortName`* port defined on the *`ServiceName`* service, Celtix generates a corresponding `get`*`PortName`*`()` method in Java. Therefore, a `wsdl:service` element that defines multiple ports will generate a service class with multiple `get`*`PortName`*`()` methods.

Service Endpoint Interface

For every port type defined in the original WSDL contract, you can generate a corresponding *service endpoint interface* in Java. A service endpoint interface is the Java mapping of a WSDL port type. Each

operation defined in the original WSDL port type maps to a corresponding method in the service endpoint interface. The operation's parameters are mapped as follows:

- The input parameters are mapped to method arguments.
- The first output parameter is mapped to a return value.
- If there is more than one output parameter, the second and subsequent output parameters map to method arguments (moreover, the values of these arguments must be passed using Holder types).

For example, Listing 3 shows the `Greeter` service endpoint interface, which is generated from the `Greeter` port type defined in Listing 1. For simplicity, Listing 3 omits the standard JAXB and JAX-WS annotations.

```
// Java
/* Generated by WSDLToJava Compiler. */

package org.objectweb.hello_world_soap_http;
...
public interface Greeter {
    public java.lang.String sayHi();

    public java.lang.String greetMe(
        java.lang.String requestType
    );

    public void greetMeOneWay(
        java.lang.String requestType
    );

    public void pingMe() throws PingMeFault;
}
```

Listing 3: The Greeter Service Endpoint Interface

Client Main Function

Listing 4 shows the Java code that implements the HelloWorld client. In summary, the client connects to the `SoapPort` port on the `SOAPService` service and then proceeds to invoke each of the operations supported by the `Greeter` port type.

```
// Java
package demo.hw.client;

import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import org.objectweb.hello_world_soap_http.Greeter;
import org.objectweb.hello_world_soap_http.PingMeFault;
import org.objectweb.hello_world_soap_http.SOAPService;

public final class Client {

    private static final QName SERVICE_NAME
        = new QName("http://objectweb.org/hello_world_soap_http", "SOAPService");


    private Client() {
    }

    public static void main(String args[]) throws Exception {

        if (args.length == 0) {
            System.out.println("please specify wsdl");
            System.exit(1);
        }
```

```
        URL wsdlURL;
        File wsdlFile = new File(args[0]);
        if (wsdlFile.exists()) {
            wsdlURL = wsdlFile.toURL();
        } else {
            wsdlURL = new URL(args[0]);
        }

        System.out.println(wsdlURL);
        SOAPService ss = new SOAPService(wsdlURL, SERVICE_NAME);
        Greeter port = ss.getSoapPort();
        String resp;

        System.out.println("Invoking sayHi...");
        resp = port.sayHi();
        System.out.println("Server responded with: " + resp);
        System.out.println();

        System.out.println("Invoking greetMe...");
        resp = port.greetMe(System.getProperty("user.name"));
        System.out.println("Server responded with: " + resp);
        System.out.println();

        System.out.println("Invoking greetMeOneWay...");
        port.greetMeOneWay(System.getProperty("user.name"));
        System.out.println("No response from server as method is OneWay");
        System.out.println();

        try {
            System.out.println("Invoking pingMe, expecting exception...");
            port.pingMe();
        } catch (PingMeFault ex) {
            System.out.println("Expected exception: PingMeFault has occurred.");
            System.out.println(ex.toString());
        }
        System.exit(0);
    }

}
```

Listing 4: Client Implementation Code

The `Client.main()` function from Listing 4 proceeds as follows:

- The Celtix runtime is implicitly initialized — that is, provided the Celtix runtime classes are loaded. Hence, there is no need to call a special function in order to initialize Celtix.
- The client expects a single string argument that gives the location of the WSDL contract for HelloWorld. The WSDL location is stored in the variable, `wsdlURL`.
- A new port object (which enables you to access the remote server endpoint) is created in two steps, as shown in the following code fragment:

```
// Java
        SOAPService ss = new SOAPService(wsdlURL, SERVICE_NAME);
        Greeter port = ss.getSoapPort();
```

  To create a new port object, you first create a service object (passing in the WSDL location and service name) and then call the appropriate `get`*`PortName`*`()` method to obtain an instance of the particular port you need. In this case, the `SOAPService` service supports only the `SoapPort` port, which is of `Greeter` type.
- The client proceeds to call each of the methods supported by the `Greeter` service endpoint interface.
- In the case of the `pingMe()` operation, the example code shows how to catch the `PingMeFault` fault

exception.

Setting Connection Properties with Contexts

You can use JAX-WS contexts to customize the properties of a client proxy. In particular, contexts can be used to modify connection properties and to send data in protocol headers. For example, you could use contexts to add a SOAP header, either to a request message or to a response message. The following types of context are supported on the client side:

- Request context — on the client side, the *request context* enables you to set properties that affect outbound messages. Request context properties are applied to a specific port instance and, once set, the properties affect every subsequent operation invocation made on the port, until such time as a property is explicitly cleared. For example, you might use a request context property to set a connection timeout or to initialize data for sending in a header.
- Response context — on the client side, you can access the *response context* to read the property values set by the inbound message from the last operation invocation. Response context properties are reset after every operation invocation. For example, you might access a response context property to read header information received from the last inbound message.

Setting a Request Context

To set a particular request context property, `ContextPropertyName`, to the value, `PropertyValue`, use the code shown in Listing 5.

```
// Java
// Set request context property.
java.util.Map<String, Object> requestContext =
    ((javax.xml.ws.BindingProvider)port).getRequestContext();
requestContext.put(ContextPropertyName, PropertyValue);

// Invoke an operation.
port.SomeOperation();
```

Listing 5: Setting a Request Context Property on the Client Side

You have to cast the port object to `javax.xml.ws.BindingProvider` type in order to access the request context. The request context itself is of type, `java.util.Map<String, Object>`, which is a hash map that has keys of `String` type and values of arbitrary type. Use the `java.util.Map.put()` method to create a new entry in the hash map.

Reading a Response Context

To retrieve a particular response context property, `ContextPropertyName`, use the code shown in Listing 6.

```
// Java
// Invoke an operation.
port.SomeOperation();

// Read response context property.
java.util.Map<String, Object> responseContext =
    ((javax.xml.ws.BindingProvider)port).getResponseContext();
PropertyType propValue = (PropertyType) responseContext.get(ContextPropertyName);
```

Listing 6: Reading a Response Context Property on the Client Side

The response context is of type, `java.util.Map<String, Object>`, which is a hash map that has keys of `String` type and values of arbitrary type. Use the `java.util.Map.get()` method to access an entry in the hash map of response context properties.

Contexts Supported by Celtix

Celtix supports the following context properties:

Context Property Name	Context Property Type
`org.objectweb.celtix.ws.addressing.JAXWSAConstants.CLIENT_ADDRESSING_PROPERTIES`	`org.objectweb.celtix.ws.addressing.AddressingProperties`

Table 1: Celtix Context Properties

[REVISIT – What is the complete list of context properties?]

[Example – maybe not included for now?]

Asynchronous Invocation Model

In addition to the usual synchronous mode of invocation, Celtix also supports two forms of asynchronous invocation, as follows:

- Polling approach — in this case, to invoke the remote operation, you call a special method that has no output parameters, but returns a `javax.xml.ws.Response` instance. The `Response` object (which inherits from the `javax.util.concurrency.Future` interface) can be polled to check whether or not a response message has arrived.
- Callback approach — in this case, to invoke the remote operation, you call another special method that takes a reference to a callback object (of `javax.xml.ws.AsyncHandler` type) as one of its parameters. Whenever the response message arrives at the client, the Celtix runtime calls back on the `AsyncHandler` object to give it the contents of the response message.

Both of these asynchronous invocation approaches are described here and illustrated by code examples.

WSDL Contract for Asynchronous Example

Listing 7 shows the WSDL contract that is used for the asynchronous example. The contract defines a single port type, `GreeterAsync`, which contains a single operation, `greetMeSometime`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://objectweb.org/hello_world_async_soap_http"
xmlns:x1="http://objectweb.org/hello_world_async_soap_http/types"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://objectweb.org/hello_world_async_soap_http" name="HelloWorld">
    <wsdl:types>
        <schema targetNamespace="http://objectweb.org/hello_world_async_soap_http/types"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:x1="http://objectweb.org/hello_world_async_soap_http/types"
elementFormDefault="qualified">
            <element name="greetMeSometime">
                <complexType>
                    <sequence>
                        <element name="requestType" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="greetMeSometimeResponse">
                <complexType>
                    <sequence>
                        <element name="responseType" type="xsd:string"/>
                    </sequence>
                </complexType>
```

```
            </element>
        </schema>
    </wsdl:types>
    <wsdl:message name="greetMeSometimeRequest">
        <wsdl:part name="in" element="x1:greetMeSometime"/>
    </wsdl:message>
    <wsdl:message name="greetMeSometimeResponse">
        <wsdl:part name="out" element="x1:greetMeSometimeResponse"/>
    </wsdl:message>

    <wsdl:portType name="GreeterAsync">
        <wsdl:operation name="greetMeSometime">
            <wsdl:input name="greetMeSometimeRequest"
message="tns:greetMeSometimeRequest"/>
            <wsdl:output name="greetMeSometimeResponse"
message="tns:greetMeSometimeResponse"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="GreeterAsync_SOAPBinding" type="tns:GreeterAsync">
        ...
    </wsdl:binding>

    <wsdl:service name="SOAPService">
        <wsdl:port name="SoapPort" binding="tns:GreeterAsync_SOAPBinding">
            <soap:address location="http://localhost:9000/SoapContext/SoapPort"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

Listing 7: HelloWorld WSDL Contract for Asynchronous Example

Generating the Asynchronous Stub Code

The asynchronous style of invocation requires extra stub code (for example, dedicated asychronous methods defined on the service endpoint interface). This special stub code is not generated by default, however. To switch on the asynchronous feature and generate the requisite stub code, you must use the mapping customization feature from the WSDL 2.0 specification.

Customization enables you to modify the way the `wsdl2java` utility generates stub code. In particular, it enables you to modify the WSDL-to-Java mapping and to switch on certain features. Here, customization is used to switch on the asynchronous invocation feature. Customizations are specified using a *binding declaration*, which you define using a `jaxws:bindings` tag (where the `jaxws` prefix is tied to the `http://java.sun.com/xml/ns/jaxws` namespace). There are two alternative ways of specifying a binding declaration:

- External binding declaration — the `jaxws:bindings` element is defined in a file separately from the WSDL contract. You specify the location of the binding declaration file to the `wsdl2java` utility when you generate the stub code.
- Embedded binding declaration — you can also embed the `jaxws:bindings` element directly in a WSDL contract, treating it as a WSDL extension. In this case, the settings in `jaxws:bindings` apply only to the immediate parent element.

This section considers only the first approach, the external binding declaration. The template for a binding declaration file that switches on asynchronous invocations is shown in Listing 8.

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

Listing 8: Template for an Asynchronous Binding Declaration

Where `AffectedWSDLContract` specifies the URL of the WSDL contract that is affected by this binding declaration. The `AffectedNode` is an XPath value that specifies which node (or nodes) from the WSDL contract are affected by this binding declaration. You can set `AffectedNode` to `wsdl:definitions`, if you want the entire WSDL contract to be affected. The `jaxws:enableAsyncMapping` element is set to `true` to enable the asynchronous invocation feature.

For example, if you want to generate asynchronous methods only for the `GreeterAsync` port type, you could specify `<bindings node="wsdl:definitions/wsdl:portType[@name='GreeterAsync']">` in the preceding binding declaration.

Assuming that the binding declaration is stored in a file, `async_binding.xml`, you can generate the requisite stub files with asynchronous support by entering the following `wsdl2java` command:

```
wsdl2java -ant -client -d ClientDir -b async_binding.xml hello_world.wsdl
```

When you run the `wsdl2java` command, you specify the location of the binding declaration file using the `-b` option. After generating the stub code in this way, the `GreeterAsync` service endpoint interface (in the file `GreeterAsync.java`) is defined as shown in Listing 9.

```
/* Generated by WSDLToJava Compiler. */

package org.objectweb.hello_world_async_soap_http;
...
import java.util.concurrent.Future;
import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;
...
public interface GreeterAsync {

    public Future<?> greetMeSometimeAsync(
        java.lang.String requestType,
        AsyncHandler<org.objectweb.hello_world_async_soap_http.types.GreetMeSometimeResp
onse> asyncHandler
    );

    public
Response<org.objectweb.hello_world_async_soap_http.types.GreetMeSometimeResponse>
greetMeSometimeAsync(
        java.lang.String requestType
    );

    public java.lang.String greetMeSometime(
        java.lang.String requestType
    );
}
```

Listing 9: Service Endpoint Interface with Methods for Asynchronous Invocations

In addition to the usual synchronous method, `greetMeSometime()`, two asynchronous methods are also generated for the `greetMeSometime` operation, as follows:

- `greetMeSometimeAsync()` method with `Future<?>` return type and an extra `javax.xml.ws.AsyncHandler` parameter — call this method for the *callback approach* to asynchronous invocation.
- `greetMeSometimeAsync()` method with `Response<GreetMeSometimeResponse>` return type — call this method for the *polling approach* to asynchronous invocation

The details of the callback approach and the polling approach are discussed in the following subsections.

Implementing an Asynchronous Client with the Polling Approach

Listing 10 illustrates the polling approach to making an asynchronous operation call. Using this approach, the client invokes the operation by calling the special Java method, *OperationName*`Async()`, that returns a `javax.xml.ws.Response<T>` object, where `T` is the type of the operation's response message. The `Response<T>` object can be polled at a later stage to check whether the operation's response message has arrived.

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.objectweb.hello_world_async_soap_http.GreeterAsync;
import org.objectweb.hello_world_async_soap_http.SOAPService;
import org.objectweb.hello_world_async_soap_http.types.GreetMeSometimeResponse;

public final class Client {

    private static final QName SERVICE_NAME
        = new QName("http://objectweb.org/hello_world_async_soap_http", "SOAPService");

    private Client() {
    }

    public static void main(String args[]) throws Exception {
        ...
        // Polling approach:
        Response<GreetMeSometimeResponse> greetMeSomeTimeResp =
            port.greetMeSometimeAsync(System.getProperty("user.name"));
        while (!greetMeSomeTimeResp.isDone()) {
            Thread.sleep(100);
        }
        GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
        ...
        System.exit(0);
    }
}
```

Listing 10: Polling Approach for an Asynchronous Operation Call

The `greetMeSometimeAsync()` method invokes the `greetMeSometimes` operation, transmitting the input parameters to the remote service and returning a reference to a `javax.xml.ws.Response<GreetMeSometimeResponse>` object. The `Response` class is defined by extending the standard `java.util.concurrency.Future<T>` interface, which is specifically designed for polling the outcome of work performed by a concurrent thread. There are essentially two basic approaches to polling using the `Response` object:

- Non-blocking polling — before attempting to get the result, check whether the response has arrived by calling the non-blocking `Response<T>.isDone()` method. For example:

```
// Java
Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;

if (greetMeSomeTimeResp.isDone()) {
    GreetMeSometimeResponse reply = greetMeSomeTimeResp.get();
}
```

- Blocking polling — call `Response<T>.get()` right away and block until the response arrives (optionally specifying a timeout). For example, to poll for a response, with a 60 second timeout:

```
// Java
Response<GreetMeSometimeResponse> greetMeSomeTimeResp = ...;

GreetMeSometimeResponse reply = greetMeSomeTimeResp.get(
                                    60L,
                                    java.util.concurrent.TimeUnit.SECONDS
                                );
```

[REVISIT – What is the general form of the method that represents the polling call? E.g. what happens with inout parameters in the method signature?]

Implementing an Asynchronous Client with the Callback Approach

An alternative approach to making an asynchronous operation invocation is to implement a callback class, by deriving from the `javax.xml.ws.AsyncHandler` interface. This callback class must implement a `handleResponse()` method, which is called by the Celtix runtime to notify the client that the response has arrived. Listing 11 shows an outline of the `AsyncHandler` interface that you need to implement.

```
// Java
package javax.xml.ws;

public interface AsyncHandler<T>
{
    void handleResponse(Response<T> res);
}
```

Listing 11: The javax.xml.ws.AsyncHandler Interface

In this example, a callback class, `TestAsyncHandler`, is defined as shown in Listing 12.

```
package demo.hw.client;

import javax.xml.ws.AsyncHandler;
import javax.xml.ws.Response;

import org.objectweb.hello_world_async_soap_http.types.GreetMeSometimeResponse;

public class TestAsyncHandler implements AsyncHandler<GreetMeSometimeResponse> {
    private GreetMeSometimeResponse reply;

    public void handleResponse(Response<GreetMeSometimeResponse> response) {
        try {
```

```
            reply = response.get();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    public String getResponse() {
        return reply.getResponseType();
    }

}
```

Listing 12: The TestAsyncHandler Callback Class

The implementation of the `handleResponse()` method shown in Listing 12 simply gets the response data and stores it in a member variable, `reply`. The extra `getResponse()` method is just a convenience method that extracts the sole output parameter (that is, `responseType`) from the response.

Listing 13 illustrates the callback approach to making an asynchronous operation call. Using this approach, the client invokes the operation by calling the special Java method, *`OperationName`*`Async()`, that returns a `java.util.concurrency.Future<?>` object and takes an extra parameter of `AsyncHandler<T>` type.

```
package demo.hw.client;

import java.io.File;
import java.util.concurrent.Future;

import javax.xml.namespace.QName;
import javax.xml.ws.Response;

import org.objectweb.hello_world_async_soap_http.GreeterAsync;
import org.objectweb.hello_world_async_soap_http.SOAPService;
import org.objectweb.hello_world_async_soap_http.types.GreetMeSometimeResponse;

public final class Client {

    private static final QName SERVICE_NAME
        = new QName("http://objectweb.org/hello_world_async_soap_http", "SOAPService");

    private Client() {
    }

    public static void main(String args[]) throws Exception {
        ...
        // Callback approach
        TestAsyncHandler testAsyncHandler = new TestAsyncHandler();
        System.out.println("Invoking greetMeSometimeAsync using callback object...");
        Future<?> response = port.greetMeSometimeAsync(System.getProperty("user.name"),
testAsyncHandler);
        while (!response.isDone()) {
            Thread.sleep(100);
        }
        resp = testAsyncHandler.getResponse();
        ...
        System.exit(0);
    }
}
```

Listing 13: Callback Approach for an Asynchronous Operation Call

The `Future<?>` object returned by `greetMeSometimeAsync()` can be used only to test whether or not a response has arrived yet — for example, by calling `response.isDone()`. The value of the response is only made available to the callback object, `testAsyncHandler`.