

# BPEL Tutorial

## Tutorial 7: Invoking BPEL Processes through SOAP and Java

As mentioned previously, a BPEL process is itself a Web service, defining and supporting a client interface via WSDL and SOAP. However, BPEL processes deployed on the Oracle BPEL Process Manager are also made available to clients via a Java API. This tutorial describes how to invoke BPEL processes, both synchronous and asynchronous, through either SOAP or Java.

### Contents

Overview .....	2
Invoking a BPEL Process with the BPEL Console .....	2
Invoking a BPEL Process with the Generic Java API .....	4
Connecting to a BPEL Process Manager with the Locator class .....	5
Passing XML messages via Java .....	5
Invoking a two-way operation via Java API .....	5
Testing invokeCreditRatingService.jsp .....	7
Invoking a one-way operation via Java API .....	8
Testing invokeHelloWorld.jsp .....	9
Retrieving Status/Results from Asynchronous BPEL Processes .....	11
Using the Java API from a Remote Client .....	12
Invoking a BPEL Process with the WebService/SOAP Interface .....	12
View the WSDL for the Deployed BPEL Process .....	13
Building, Deploying and Testing the SOAP Client .....	14
SOAP Request Content .....	17
Review the Implementation of the Axis Client .....	18
Creating a BPEL Process Web Service Client “from Scratch” with Axis .....	19
Receiving Asynchronous Callbacks via SOAP .....	21

## Overview

In previous tutorials, you have tested your newly created BPEL processes with the BPEL Console. This console is useful for testing purposes during development but ultimately you will need to use an API or develop a custom GUI to initiate BPEL processes.

A BPEL process deployed to the Oracle BPEL Process Manager can be invoked through three mechanisms:

1. The BPEL Console's Initiate tab (where you can specify XML data to pass into the process as input or use the automatically generated HTML test form interface).
2. Through the generic Java API published by the Oracle BPEL Process Manager.
3. Through its Web service/SOAP interface.

This tutorial examines how each of these interfaces may be used to create instances of deployed processes.

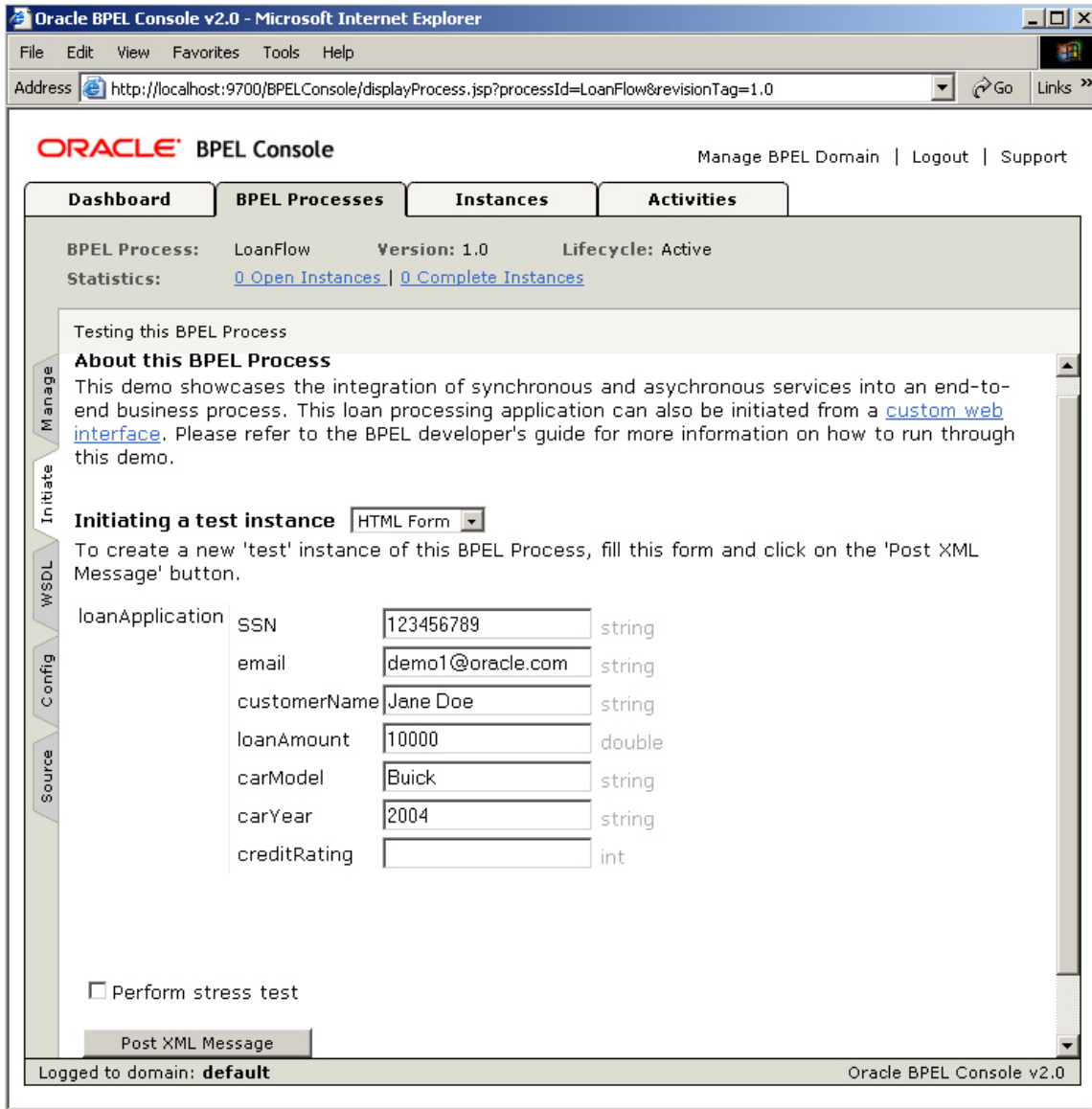
## Invoking a BPEL Process with the BPEL Console

In prior tutorials you have seen how to invoke a BPEL process using the BPEL Console so we do not discuss it in great detail here. One thing to note, however, is that the BPEL Console supports two modes of invocation: an HTML Form view and an XML Source mode.

The XML Source mode, shown below, allows you to enter XML source data directly into a text form and that source is passed, as is, to initiate the BPEL process. To simplify the use of this mode, you can configure default XML data in the deployment descriptor of the process, which will be pre-loaded into the XML data field (see the `defaultInput` property in the file `C:\orabpel\samples\demos\LoanDemo\LoanFlow\bpel.xml` for an example of how to do this).



Alternatively, the HTML Form view in the BPEL Console's Initiate tab will automatically generate an HTML input form (when possible), which can be used to provide the input values to initiate a BPEL process for testing purposes:



But, as mentioned, the goal of this tutorial is not to describe testing BPEL processes through the console, so we won't consider it further.

## Invoking a BPEL Process with the Generic Java API

Processes can be invoked programmatically via a Java API provided through a stateless session bean interface by the BPEL Process Manager. The API is slightly different depending on whether you are invoking a two-way operation (which has both input and output messages) or a one-way operation (which just has an input message and returns no result). As such, two code examples are provided in the BPEL Process Manager samples directory, one for invoking the CreditRating BPEL process, which provides a synchronous service with a two-way `process` operation, and one for initiating the HelloWorld BPEL process, which is an asynchronous service with a one-way

initiate operation. These two samples are discussed below, but both use a few building blocks that we will first describe here.

## Connecting to the Oracle BPEL Process Manager with the Locator class

To support a flexible client interface without being affected by server clustering and other production configuration details, a **com.oracle.bpel.client.Locator** class is provided which is used to connect to a BPEL Process Manager, authenticate if required, and then get handles to services provided by that server. For example, the Locator class could be used to connect to the default domain on a local BPEL Process Manager and fetch a list of BPEL processes deployed on that server. In this case, we use the Locator class to return a handle to an **com.oracle.bpel.client.dispatch.IDeliveryService** instance which can be used to invoke/initiate BPEL processes deployed on a BPEL Process Manager:

```
import com.oracle.bpel.client.Locator;
import com.oracle.bpel.client.dispatch.IDeliveryService;

// Connect to domain "default" using password "bpel"
// null IP address means local server

Locator locator = new Locator("default", "bpel", null);

IDeliveryService deliveryService =
    (IDeliveryService)locator.lookupService
        (IDeliveryService.SERVICE_NAME );
```

## Passing XML messages via Java

Because all Web services, including BPEL processes, accept and return XML messages, any Java API which will invoke those Web services needs to leverage a way to pass XML data via Java. Going with a very simple approach, which allows support for any and all XML documents and XML Schema types, the Oracle BPEL Process Manager has a client class, **com.oracle.bpel.client.NormalizedMessage**, which allows the developer to construct an XML message dynamically. For example, to construct an input message for the CreditRatingService from static string XML data, you could use the code:

```
import com.oracle.bpel.client.NormalizedMessage;

String xml =
    "<ssn xmlns=\"http://services.otn.com\">123456789</ssn>";

NormalizedMessage nm = new NormalizedMessage( );
nm.addPart("payload", xml );
```

In practice, of course, you would construct NormalizedMessages much more dynamically. For full documentation of the NormalizedMessage class, see the Oracle BPEL Process manager JavaDocs in:

C:\orabpel\docs\apidocs

## Invoking a two-way operation via Java API

Once a delivery service has been instantiated, it can be used to initiate the BPEL process with a NormalizedMessage XML message. If you will be invoking a two-way Web

service operation which will return a result synchronously, then you will want to use one of the `IDeliveryService.request()` methods. This method is overloaded and you should see the JavaDoc for all the available versions of it, however here we will use the `request()` method with the following signature:

```
public NormalizedMessage request(java.lang.String processId,  
                                java.lang.String operationName,  
                                NormalizedMessage message)  
    throws java.rmi.RemoteException
```

A code example of using this API to invoke the `CreditRatingService` BPEL process is provided with the Oracle BPEL Process Manager samples and is shown below.

### Sample location:

C:\orabpel\samples\tutorials\102.InvokingProcesses\jsp\invokeCreditRatingService.jsp

### Full JSP source:

```
<%@page import="java.util.Map" %>  
<%@page import="com.oracle.bpel.client.Locator" %>  
<%@page import="com.oracle.bpel.client.NormalizedMessage" %>  
<%@page import="com.oracle.bpel.client.dispatch.IDeliveryService" %>  
  
<html>  
  
<head>  
  
<title>Invoke CreditRatingService</title>  
  
</head>  
  
<body>  
  
<%  
    String ssn = request.getParameter("ssn");  
    if(ssn == null)  
        ssn = "123-12-1234";  
  
    String xml = "<ssn xmlns=\"http://services.otn.com\">"  
                + ssn + "</ssn>";  
  
    Locator locator = new Locator("default","bpel",null);  
  
    IDeliveryService deliveryService =  
        (IDeliveryService)locator.lookupService  
            (IDeliveryService.SERVICE_NAME );  
  
    // construct the normalized message and send to oracle bpel process  
manager  
    NormalizedMessage nm = new NormalizedMessage( );  
    nm.addPart("payload", xml );  
  
    NormalizedMessage res =  
        deliveryService.request("CreditRatingService", "process", nm);  
  
    Map payload = res.getPayload();
```

```
out.println( "BPELProcess CreditRatingService executed!<br>" );  
out.println( "Credit Rating is " + payload.get("payload") );
```

%>

## Testing invokeCreditRatingService.jsp

- 1 Make sure that you have the CreditRatingService BPEL process deployed to your local Oracle BPEL Process Manager.

```
> cd C:\orabpel\samples\utils\CreditRatingService
```

```
> obant
```

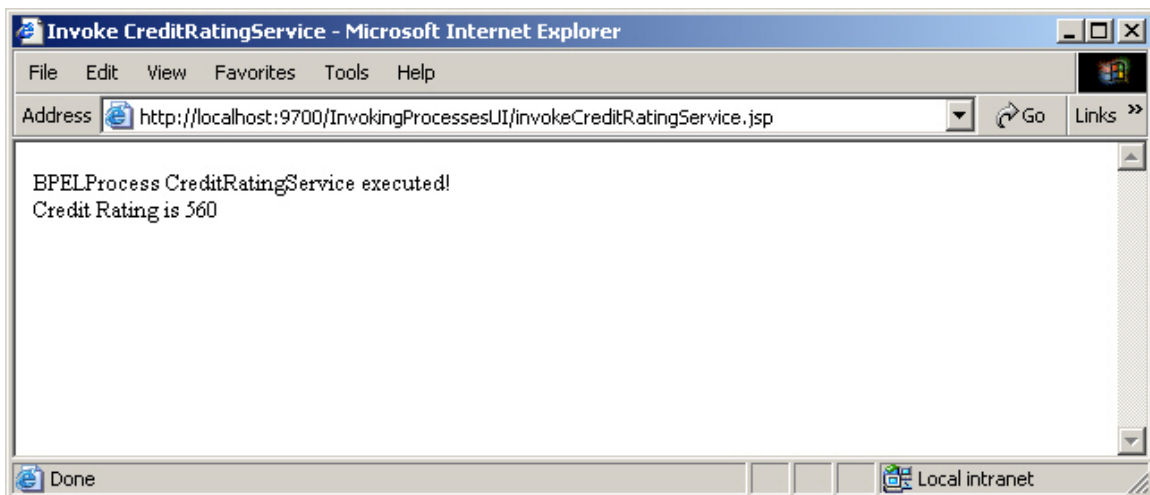
- 2 Deploy the JSP with the following commands:

```
> cd C:\orabpel\samples\tutorials\102.InvokingProcesses
```

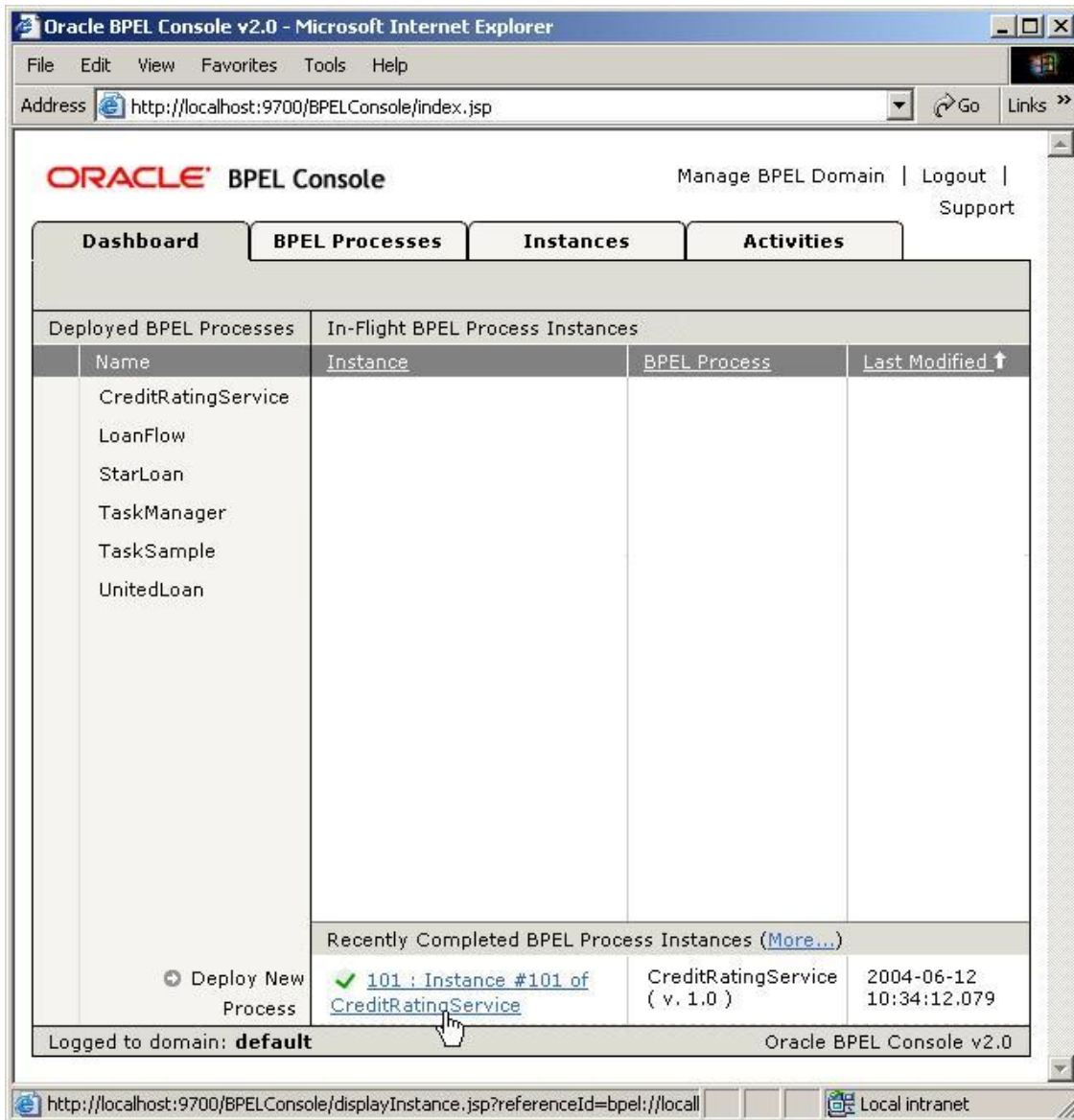
```
> obant
```

- 3 Point a browser at the URL:

<http://localhost:9700/InvokingProcessesUI/invokeCreditRatingService.jsp>



- If you want, you can now connect to your BPEL Console to see that a new instance of the CreditRatingService has been created and completed.



### Invoking a one-way operation via Java API

The procedure for invoking a one-way BPEL operation via the Java API is very similar, except that you will use the `IDeliveryService.post()` method (which is also overloaded). These methods invoke a one-way operation on a BPEL process and thus return void since a response is not expected (at least not a synchronous response).

In the code example shown here, the post method used is exactly the same as the `request()` shown above, except that it returns void:

From JavaDoc for **`com.oracle.bpel.client.dispatch.IDeliveryService`**:



```
public void post(java.lang.String processId,  
                java.lang.String operationName,  
                NormalizedMessage message)  
    throws java.rmi.RemoteException
```

### Sample location:

C:\orabpel\samples\tutorials\102.InvokingProcesses\jsp\invokeHelloWorld.jsp

### Selected JSP source:

```
<%@page import="com.oracle.bpel.client.Locator" %>  
<%@page import="com.oracle.bpel.client.NormalizedMessage" %>  
<%@page import="com.oracle.bpel.client.dispatch.IDeliveryService" %>  
...  
    Locator locator = new Locator("default", "bpel", null);  
...  
    NormalizedMessage nm = new NormalizedMessage( );  
    nm.addPart("payload" , xml );  
    deliveryService.post("HelloWorld", "initiate", nm);  
    out.println( "BPELProcess HelloWorld initiated!" );  
<%>
```

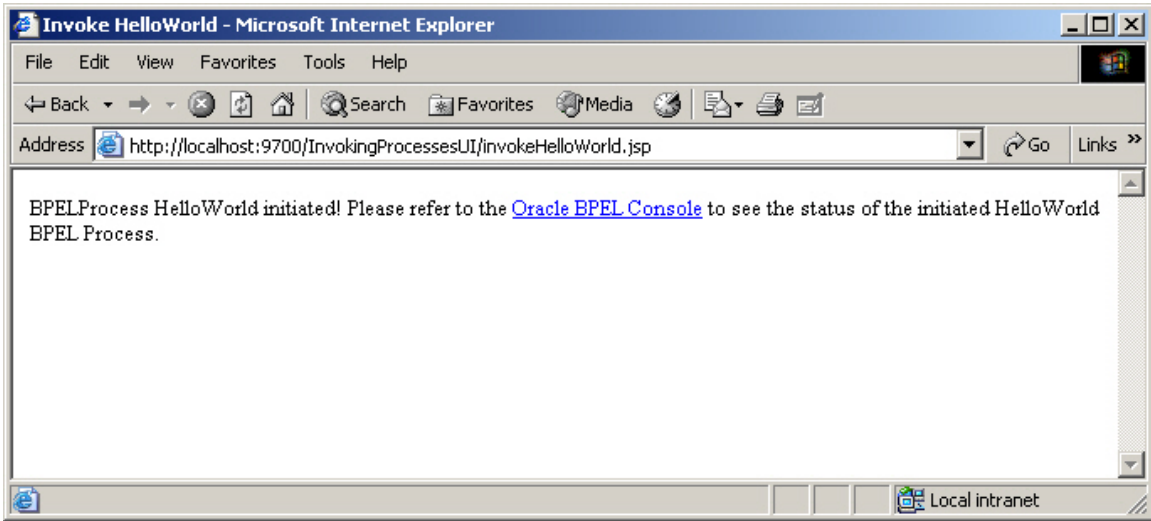
### Testing invokeHelloWorld.jsp

- 5 Make sure that you have the HelloWorld BPEL process deployed to your local Oracle BPEL Process Manager:  

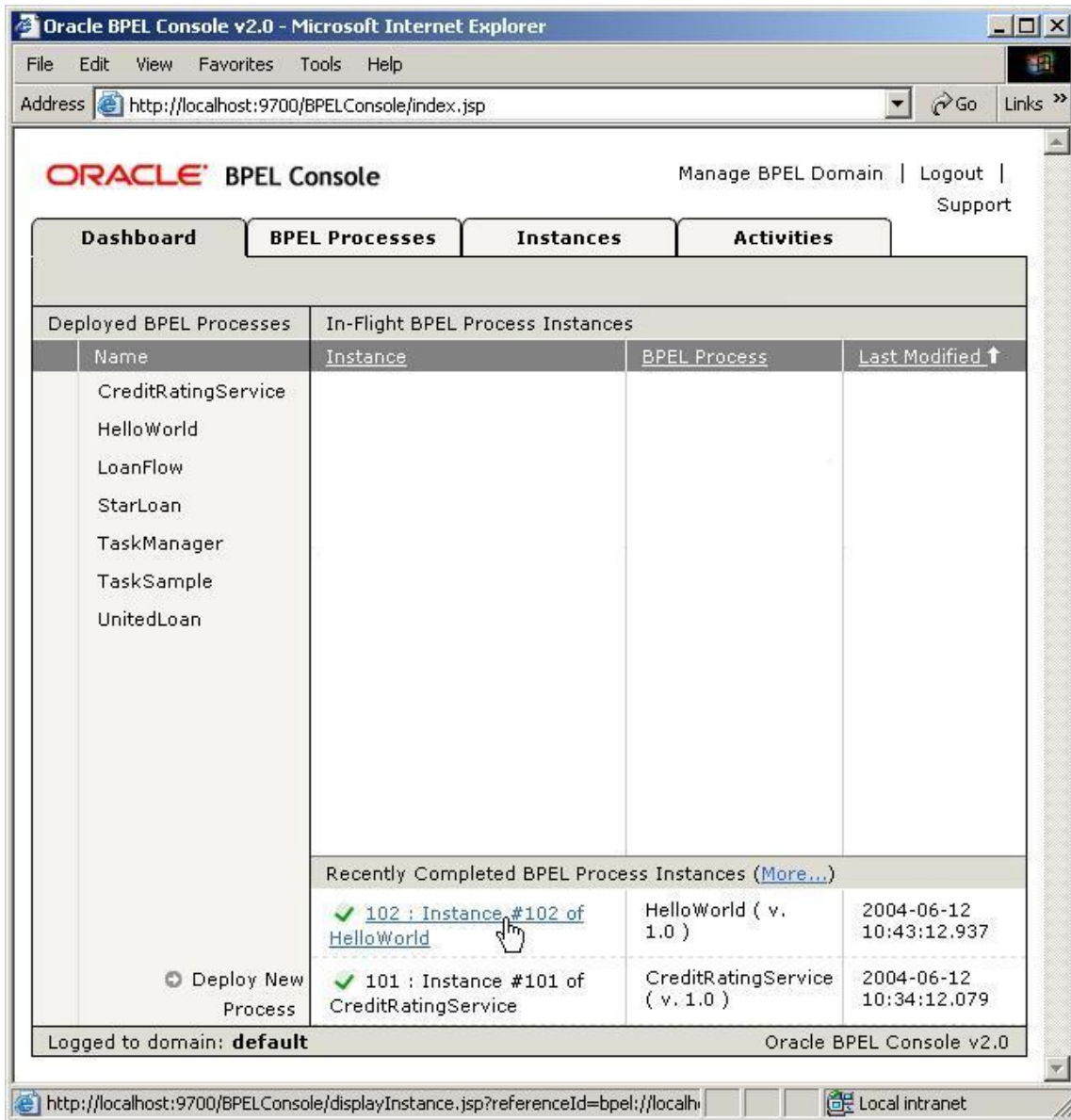
```
> cd C:\orabpel\samples\tutorials\101.HelloWorld  
> obant
```
- 6 Deploy the JSP with the following commands (if you didn't do it already for the JSP which invoked the \CreditRatingService):  

```
> cd C:\ orabpel \samples\tutorials\102.InvokingProcesses  
> obant
```

- 7 Point a browser at the URL:  
<http://localhost:9700/InvokingProcessesUI/invokeHelloWorld.jsp>



- 8 Again, you could now connect to the BPEL Console to see that a new instance of the HelloWorld process has now been created.



## Retrieving Status/Results from Asynchronous BPEL Processes

If you use the Java API to initiate an asynchronous BPEL process, then you often have to consider how to receive the result of the process since a Java client cannot be called back the same way a Web service can be. Of course, in some cases, this is not an issue. For example, the LoanFlowPlus BPEL demo application (located in C:\orabpel\samples\demos\LoanDemoPlus) avoids this issue by informing users of process progress through a user task where the user can manually approve the final loan offer. In some cases, the process will send some sort of notification, such as an email message or a JMS message, when it completes – see the samples

C:\orabpel\samples\tutorials\116.SendEmails and  
C:\orabpel\samples\tutorials\118.JMSService\buyer for examples of how to send email or  
JMS messages from BPEL processes.

Also, a Java client can poll for the result of an asynchronous BPEL process. In this case, the client will need a handle to be able to fetch status information for a particular instance. And while the `post()` method does not automatically return such a handle, it does support the client specifying a “conversation ID” which can be any unique identifier that the client can later use to identify a specific instance and retrieve status information for it. See the JavaDocs for the **`com.oracle.bpel.client.NormalizedMessage`** class to see the specific field name for the conversation ID and other properties, which can be set at the time a BPEL process, is instantiated via the Java API. And also see the **`com.oracle.bpel.client.Locator.lookupInstance(String key)`** method to be able to locate a specific instance based on a conversation ID.

Finally it is possible using the supported `NormalizedMessage` properties to even specify the address of a Web service for the callback and therefore to initiate an asynchronous BPEL process from Java, but receive a SOAP/XML callback to a Web service listener. This is a more advanced use-case, so you should contact your Oracle support representative for more information on how to accomplish this in your specific environment.

### Using the Java API from a Remote Client

The code examples as described above are executed within the same application server container as the BPEL Process Manager is running in. These APIs are remotable, however, and can be used via RMI from a remote application server. We do not currently ship code examples for this use-case - in part because the RMI client code is different based on which application server the client is running in. You should work with your Oracle support representative regarding how to use the BPEL Process Manager Java API over RMI for your specific client configuration/environment.

### Invoking a BPEL Process with the WebService/SOAP Interface

Once deployed to an Oracle BPEL Process Manager, a BPEL process is automatically published as a Web service. This means that the process can be accessed via its XML/SOAP/WSDL interface without any additional developer effort.

Supporting a standard Web services interface means that BPEL processes can be invoked from any client technology which supports Web services – including Microsoft .NET™, Sun’s JAX-RPC implementation, Apache Axis, Oracle JDeveloper and the many other Web services toolkits available. In addition it means that BPEL and the Oracle BPEL Process Manager can be used to publish Web services and those services, both synchronous and asynchronous, can be invoked from applications and services implemented with nearly any technology and language.

You access a BPEL process through its Web service interface in the standard way you would access any Web service – by writing a client which uses the BPEL process’ WSDL interface definition and SOAP as a protocol.

Here we will describe how to invoke a BPEL process from a Web service client developed with the open source Apache Axis toolkit. For other Web service toolkits, you should consult the documentation for that toolkit and/or contact your Oracle representative.

These examples use the UseStockReviewSheet BPEL process as a building block so you should make sure that this process has been deployed to your server using the following steps.

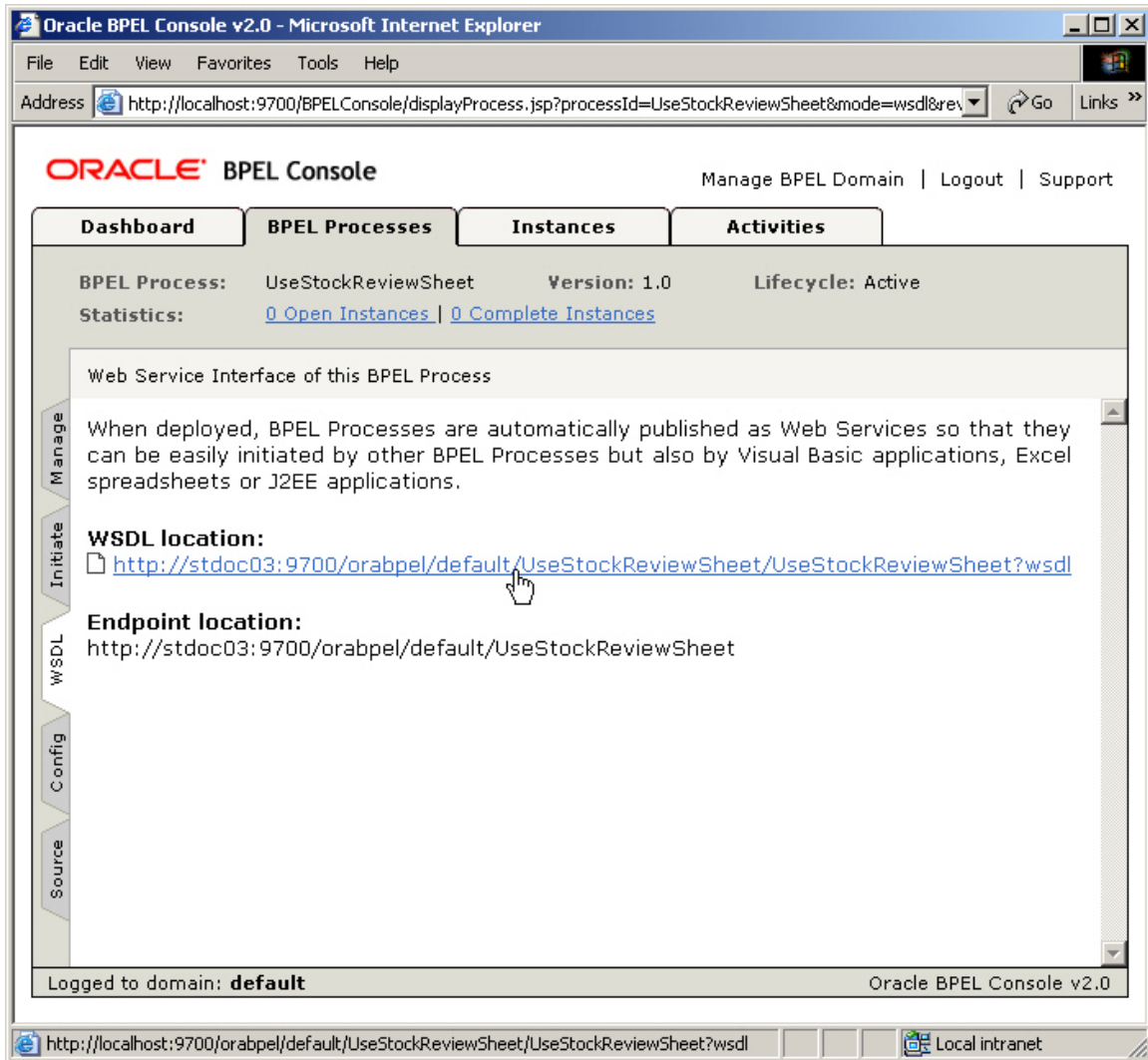
**To compile and deploy the UseStockReviewSheet process from the command-line:**

- 1 Open up a command prompt if you do not already have one open.
- 2 Compile and deploy the UseStockReviewSheet process as follows:
  - > **cd C:\orabpel\samples\tutorials\103.XMLDocuments**
  - > **obant**

**View the WSDL for the Deployed BPEL Process**

The Oracle BPEL Process Manager automatically adds binding information to the WSDL for a BPEL process and publishes it at a standard location. In addition, the BPEL Process Manager provides a SOAP listener endpoint for the deployed BPEL processes so they can be invoked through SOAP from any Web service client.

Once you have deployed a BPEL process, you can find the location of its WSDL by connecting to the BPEL Console, selecting the process and then clicking the WSDL tab. For example, for the UseStockReviewSheet process:



## Building, Deploying and Testing the SOAP Client

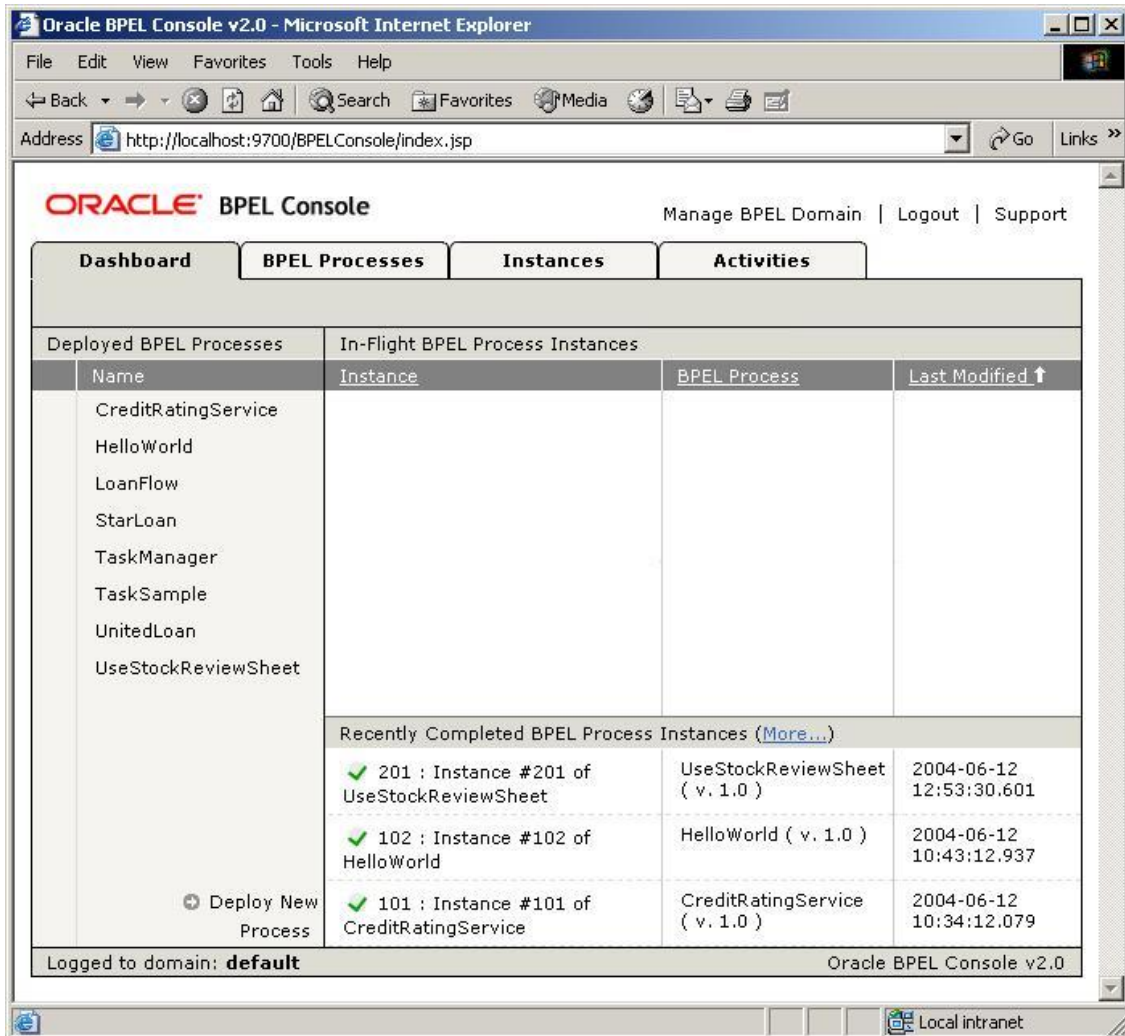
An example of a SOAP client, which invokes (initiates) the UseStockReviewSheet BPEL process, is provided with the Oracle BPEL Process Manager samples. As mentioned, this sample is built with the open source Apache Axis toolkit (which provides a fairly mature and commercially viable JAX-RPC implementation), however it should be possible to accomplish the same thing with any Web service-enabled technology. Also keep in mind that this client just initiates an asynchronous BPEL process and does not receive its resulting callback. This issue is discussed further in the next section of this tutorial.

**Sample location:** C:\orabpel\samples\tutorials\102.InvokingProcesses\ws

Follow the steps below to invoke and test the Axis client for the UseStockReviewSheet BPEL process:

- 1 You will need to have Apache Axis and Ant installed somewhere on your system. These instructions have been tested with the 1.1 Axis release (<http://ws.apache.org/axis/>) and Jakarta Ant release 1.5.1 (<http://archive.apache.org/dist/ant/binaries/>).
- 2 Open up the file  
C:\orabpel\samples\tutorials\102.InvokingProcesses\ws\setenv.bat (or setenv.sh for Unix/Linux) in a text editor and fill in the correct locations for the AXIS\_HOME and ANT\_HOME variables.
- 3 Run ant.cmd (or ant.sh) as shown below to compile the AXIS Java client classes  
> **cd C:\orabpel\samples\tutorials\102.InvokingProcesses\ws**  
> **ant.cmd**
- 4 Make sure your BPEL Process Manager is running and execute the runUseStockReviewSheetClient.cmd (.sh) script to run the Axis client and invoke the BPEL process via SOAP.  
> **runUseStockReviewSheetClient.cmd**

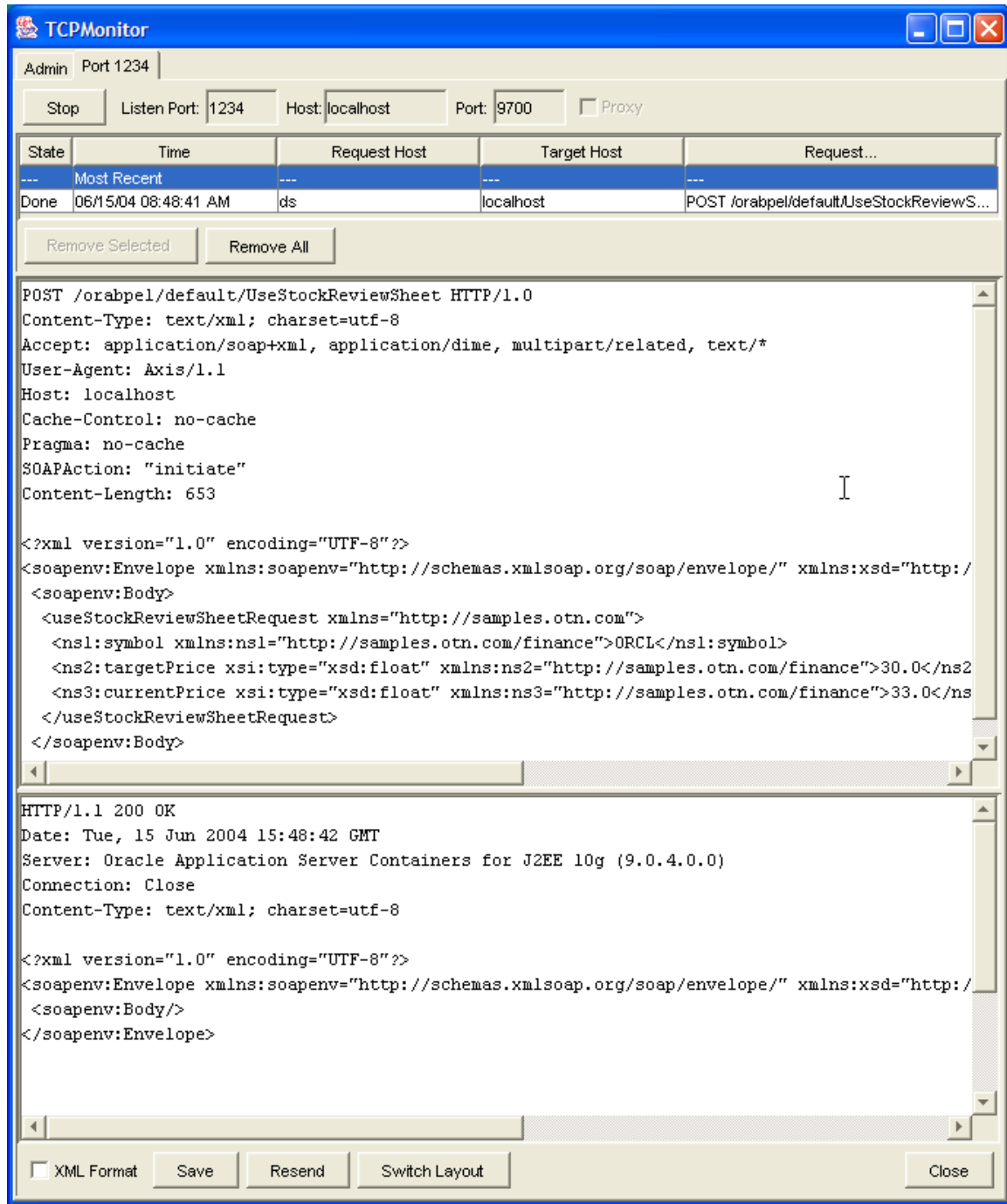
- You can now connect to the BPEL Console to see that a new instance of the BPEL process has been created (and in this case completed, since this process runs to completion fairly quickly)





## SOAP Request Content

If you use a TCP Tunnel or other tool for viewing the on-the-wire content of the SOAP request sent by the Axis client to the BPEL Process Manager, it would look as follows. For more information on TCP tunneling with the Oracle BPEL Process Manager, see the technote available at <http://otn.oracle.com/bpel> (note that to change the endpoint for the client to be your TCP tunnel, you would edit the `runUseStockReviewSheet.cmd` script).



The screenshot shows the TCPMonitor application window. At the top, there are controls for 'Admin' (Port 1234), 'Stop', 'Listen Port' (1234), 'Host' (localhost), 'Port' (9700), and a 'Proxy' checkbox. Below this is a table with columns: State, Time, Request Host, Target Host, and Request... The table contains one entry: State: Done, Time: 06/15/04 08:48:41 AM, Request Host: ds, Target Host: localhost, Request: POST /orabpel/default/UseStockReviewS... Below the table are 'Remove Selected' and 'Remove All' buttons. The main area is a text editor showing the raw HTTP and SOAP content. The request part shows headers like 'Content-Type: text/xml; charset=utf-8', 'Accept: application/soap+xml, application/dime, multipart/related, text/\*', 'User-Agent: Axis/1.1', 'Host: localhost', 'Cache-Control: no-cache', 'Pragma: no-cache', 'SOAPAction: "/>

## Review the Implementation of the Axis Client

Axis supports two mechanisms for writing Web service/SOAP clients:

- 1) Coding against Axis API “by hand”
- 2) The `wsdl2java` tool which will generate client stubs for a specific service’s WSDL

The `UseStockReviewSheet` process client was implemented with the first approach (which results in somewhat cleaner code). While not intended to be a complete Axis tutorial, here we review this implementation quickly. The next section provides a tutorial for using the Axis `wsdl2java` tool to build a BPEL process client from scratch.

Files in `C:\orabpel\samples\tutorials\102.InvokingProcesses\ws:`

- `setenv.bat [.sh]`: Sets up system-dependent environment for compiling and running Axis client
- `build.xml`: Ant build script for compiling Axis client
- `ant.cmd [.sh]`: Just a wrapper around Ant which sets up the environment (per `setenv.bat`) and then invokes the standard Ant build tool
- `runUseStockReviewSheet.cmd [.sh]`: A simple script which sets up the environment and then instantiates the Java client which invokes the BPEL process
- `src\com\otn\samples\UseStockReviewSheetClient.java`: Source code for Java class which uses the Apache API to invoke the `UseStockReviewSheet` BPEL process as a Web service

The most pertinent code from the `UseStockReviewSheetClient.java` implementation is shown here:

```
import javax.xml.rpc.ServiceFactory;
import org.apache.axis.client.Call;
...

SERVICE_NAME = new QName(THIS_NAMESPACE, "UseStockReviewSheet");
PORT_TYPE = new QName(THIS_NAMESPACE, "UseStockReviewSheet") ;
OPERATION_NAME =
    new QName(THIS_NAMESPACE, "useStockReviewSheetRequest");
SOAP_ACTION = "initiate";
STYLE = "wrapped";
...

public void initiate(String symbol)
{
    try
    {
        /* Create Service and Call object */
        /*****
        ServiceFactory serviceFactory = ServiceFactory.newInstance();

        Service service = serviceFactory.createService( SERVICE_NAME );

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

```

/* Set all of the stuff that would normally come from WSDL */
/*****
call.setTargetEndpointAddress( location );

call.setProperty(Call.SOAPACTION_USE_PROPERTY, Boolean.TRUE);
call.setProperty(Call.SOAPACTION_URI_PROPERTY, SOAP_ACTION);

call.setProperty( Call.OPERATION_STYLE_PROPERTY , STYLE );

call.setOperationName(OPERATION_NAME);

call.addParameter(new QName(PARAMETER_NAMESPACE, "symbol"),
    XMLType.XSD_STRING, ParameterMode.IN);
call.addParameter(new QName(PARAMETER_NAMESPACE, "targetPrice"),
    XMLType.XSD_FLOAT, ParameterMode.IN);
call.addParameter(new QName(PARAMETER_NAMESPACE, "currentPrice"),
    XMLType.XSD_FLOAT, ParameterMode.IN);

Object [] params = new Object [3];

params[0] = symbol;
params[1] = new Float(30.0f);
params[2] = new Float(33.0f);

/* Invoke the service */

call.invokeOneWay(params);

System.out.println(
    "UseStockReviewSheet BPEL process initiated");
}
catch (SOAPFaultException e)
// ... Handle exceptions ...
}
}

```

This code is written using as much of the generic JAX-RPC API as possible, minimizing the effort to port it to other JAX-RPC implementations. On the other hand, writing clients like this requires a decent understanding of the Apache Axis APIs as well as the ability to understand the WSDL for the service. For example, the invocation style above (“wrapped”) may not be obvious to developers who do not have a deep understanding of WSDL. This is exactly why tools like `wsdl2java` have been created which will generate stubs for invoking services like the above based on automated parsing of a service WSDL. In the next section, we describe how to use `wsdl2java` to automatically generate Web service client code for a BPEL process.

### Creating a BPEL Process Web Service Client “from Scratch” with Axis

Here we describe how to use the Axis `wsdl2java` tool to generate Web service client stubs to invoke the LoanFlowPlus BPEL process, followed by example of a Java class which uses the stubs.

- 1 Make sure the LoanFlowPlus process is deployed to your local BPEL Process Manager:

```
> cd C:\orabel\samples\demos\LoanDemoPlus
```

> obant

- 2 Add your Axis jars to your CLASSPATH

> set AXIS\_HOME= <wherever you installed Axis>

> set ANT\_HOME= <wherever you installed Ant>

> set

```
CLASSPATH=%CLASSPATH%;%AXIS_HOME%\lib\axis.jar;%AXIS_
HOME%\lib\wsdl4j.jar; %AXIS_HOME%\lib\commonslogging.jar;
%AXIS_HOME%\lib\commons-
discovery.jar;%AXIS_HOME%\lib\jaxrpc.jar;
%AXIS_HOME%\lib\saaj.jar
```

- 3 Create a new directory <somewhere> and invoke the wsdl2java tool from there

> java org.apache.axis.wsdl.WSDL2Java

**http://localhost:9700/orabpel/default/LoanFlowPlus/LoanFlowPlus?wsdl**

This will generate the Axis client stubs as Java classes and also create Java beans for serialization and deserialization when possible.

You can now use the Axis User's Guide docs on the `wsdl2java` tool to see how to write a Java client which uses the stubs generated by `wsdl2java`

(<http://ws.apache.org/axis/java/user-guide.html>)

- 4 Create a Java class for invoking the generated stub classes. The code shown below presumes you create a class named "Tester.java" and is based on the sample Tester.java code described in the Apache Axis User's Guide's `wsdl2java` section.

```
import com.otn.samples.*;
import com.autoloan.www.ns.autoloan.*;

public class Tester
{
    public static void main(String [] args) throws Exception {
        // Make a service
        LoanFlowPlus_Service service =
            new LoanFlowPlus_ServiceLocator();

        // Use the service to get a stub which implements the SDI.
        LoanFlowPlus_Port port = service.getLoanFlowPlusPort();

        // Set up the object which will be the input message
        // Note that this Java bean is generated by wsdl2java
        LoanApplicationType loanApplication =
            new LoanApplicationType( );

        loanApplication.setSSN("123456789");
        loanApplication.setEmail("demo1@otn.com");
        loanApplication.setCustomerName("Jane Doe");
        loanApplication.setLoanAmount(10000);
        loanApplication.setCarModel("Buick");
        loanApplication.setCarYear("1968");

        // Make the actual call
        port.initiate(loanApplication);
    }
}
```

```

        System.out.println(
            "LoanFlowPlus BPEL process initiated!");
    }
}

```

- 5 Compile your Java tester class along with the generated Java stubs. You could create a simple Ant build script to make this easy, placing the below in a build.xml file in the same directory as where you ran wsdl2java:

```

<?xml version="1.0"?>
<project name="InvokingWSClient" default="main" basedir=".">

    <target name="main">
        <mkdir dir="${basedir}/classes"/>
        <javac srcdir="${basedir}" destdir="${basedir}/classes"/>
    </target>

</project>

```

And then invoke:

```
> %ANT_HOME%\bin\ant.cmd
```

- 6 Finally, you can initiate the BPEL process with:

```
> java -classpath "./classes;%CLASSPATH%" Tester
```

Note that wsdl2java will actually generate some extra classes here in com\cxdn\samples for the callback portType. These classes are not really useful but there is no way to tell wsdl2java which portTypes to create stubs for so you could just delete them manually. In particular, they would **not** be helpful to implement an asynchronous callback listener, as is described in more detail in the following section.

## Receiving Asynchronous Callbacks via SOAP

Both of the Web service client examples above illustrate how to initiate an asynchronous BPEL process via its SOAP/Web services interface. However, additional work is required if you want to be able to implement a SOAP endpoint which listens for a callback from the BPEL process. This may not be required, as was discussed in the section on invoking BPEL processes from Java, but if it is then two main steps must be followed:

- 1) WS-Addressing information must be passed in the SOAP header request to indicate correlation information and the address of the callback endpoint.
- 2) A listener must be deployed at the client end which implements the callback interface.

A sample is provided illustrating how to accomplish this with the Apache Axis toolkit in:

```
C:\orabpel\samples\interop\axis\AXISCallingAsyncBPEL
```

For more information regarding this functionality or to accomplish this with other Web service toolkit clients, please contact your Oracle BPEL Process Manager support representative or <http://otn.oracle.com/bpel>.