Developing JAX-WS web service integration solutions using WebSphere Integration Developer V7

Bhargav Perepa

July 21, 2010

This article describes how to design, develop and test integration solutions featuring business process, mediation and JAX-WS web service components. We'll explore a variety of component logic invocation models and solution composition design patterns with an example use case developed using WebSphere Integration Developer and tested with WebSphere Process Server.

Introduction

WebSphere Integration Developer (hereafter called Integration Developer) is an Eclipse-based integration development environment (IDE) tool that facilitates the building of end-to-end service oriented architecture (SOA) integration solutions consisting of Web services, Enterprise Information System (EIS) services, workflow and mediation flow applications.

WebSphere Process Server (hereafter called Process Server) is an SOA runtime platform that features a uniform invocation programming model of Service Component Architecture (SCA), a uniform data representation model of Service Data Objects (SDO) and a Common Event Infrastructure (CEI) to generate events for monitoring and managing applications that run on it.

ANSI NIST-CSL 1-1993 (Data Format for the Interchange of Fingerprint Information) and ANSI/NIST-ITL 1a-1997 (Data Format for the Interchange of Fingerprint, Facial & SMT Information) standards define electronic encoding and transmitting of fingerprint image, identification, and arrest data between various federal, state, county and local law enforcement agencies. These ANSI standards define the content, format and units of measurement for the exchange of information related to fingerprint identification of an individual. The process of collecting all ten fingerprint scans (one from each finger of the right and left hands of an individual) is known as ten-print collection. There are several types of electronic submissions (request or response types), several types of transaction codes and their corresponding descriptions.

Table 1 and Table 2 summarize the request and response types of ten-print electronic submissions:

Table 1. Ten-print electronic submission - request type

<table>
<thead>
<tr>
<th>Type of transaction (TOT)</th>
<th>Transaction description</th>
</tr>
</thead>
</table>

Table 2. Ten-print electronic submission - response type

<table>
<thead>
<tr>
<th>Type of transaction (TOT)</th>
<th>Transaction description</th>
</tr>
</thead>
</table>
CAR  Criminal Ten-Print Submission (Answer Required)
CNA  Criminal Ten-Print Submission (No Answer Necessary)
FANC  Federal Applicant (No Charge)
FAUF  Federal Applicant User Fee
NFAP  Non-Federal Advanced Payment
NFUF  Non-Federal Applicant User Fee
MAP  Miscellaneous Applicant Civil
DEK  Known Deceased
DEU  Unknown Deceased
MPR  Missing Person
AMN  Amnesia Victim

Table 2. Ten-print electronic submission – response type

<table>
<thead>
<tr>
<th>Type of transaction (TOT)</th>
<th>Transaction description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRE</td>
<td>Submission Results - Electronic</td>
</tr>
<tr>
<td>ERRT</td>
<td>Ten-Print Transaction Error</td>
</tr>
</tbody>
</table>

Let's consider a simple use case of a web service look-up of a description, given the transaction type code and submission type for a ten-print electronic submission. Let's encapsulate a typical ten-print electronic submission by Java™ class `TenPrintSubmission`. This class contains three protected attributes and six methods, as shown in Listing 1.

Listing 1. TenPrintSubmission.java

```java
public class TenPrintSubmission implements Serializable {
    ..
    protected String submissionType;
    protected String transactionType;
    protected String transactionDescription;

    public String getSubmissionType() {
        return submissionType;
    }

    public void setSubmissionType(String value) {
        this.submissionType = value;
    }

    public String getTransactionType() {
        return transactionType;
    }

    public void setTransactionType(String value) {
        this.transactionType = value;
    }

    public String getTransactionDescription() {
        return transactionDescription;
    }

    public void setTransactionDescription(String value) {
        this.transactionDescription = value;
    }

    ..
}
```
Now let’s encapsulate our description look-up function by another Java class DescriptionLookup. This class builds a hash table of key-value pairs at the class instantiation. The getDescription(..) method performs the description look-up and initializes the description attribute value of ten-print electronic submission.

Listing 2. DescriptionLookup.java

```java
public class DescriptionLookup implements Serializable {

    static final long serialVersionUID = 1L;
    String[] Keys = {"CAR","CNA","FANC","FAUF","NFAP","NFUF","MAP","DEK","DEU","MPR",
    "AMN","SRE","ERRT");
    String[] Values = {"Criminal Ten-Print Submission
    (Answer Required)",
    "Criminal Ten-Print Submission (No Answer Necessary)",
    "Federal Applicant (No Charge)",
    "Federal Applicant User Fee", "Non-Federal Advanced
    Payment", "Non-Federal Applicant User Fee", "Miscellaneous
    Applicant Civil", "Known Deceased", "Unknown Deceased",
    "Missing Person", "Amnesia Victim", "Submission Results
    - Electronic ", "Ten-Print Transaction Error"};

    Hashtable kvHashTable = new Hashtable();

    public DescriptionLookup() {
        for (int i = 0, n = Keys.length; i < n; i++) {
            kvHashTable.put(Keys[i], new String(Values[i]));
        }
    }

    public TenPrintSubmission getDescription(TenPrintSubmission input) {
        String description = ((String)kvHashTable.get(input.getTransactionType()
        .toUpperCase()));
        if (description != null) {
            input.setTransactionDescription(description);
        } else {
            input.setTransactionDescription("Unknown Transaction Type");
        }

        return input;
    }
}
```

Build a JAX-WS web service and client

Using the two simple Java classes above, you can build a bottom-up JAX-WS web service and service client using Integration Developer V7, as shown in Figure 1.

Figure 1. JAX-WS description lookup service and web service client

To do this, complete the following steps:
1. Open the Integration Developer workbench.
2. Open the Java perspective by selecting **Window => Open Perspective => Other => Java**, and clicking **OK**, as show in Figure 2.

**Figure 2. Open Java perspective**

![Open Perspective](image)

3. Create a new Java project to hold the Java programming artifacts by doing the following:
   1. **Click File** => **New** => **Java Project** and specify the **Project name** as **DescriptionLookup**, as shown in Figure 3, and click **Next**.
2. Click **Finish** on the Java Settings screen.

4. To create a Java package for your programming artifacts and use cases, click **File => New => Package**. As shown in Figure 4, specify the **Source folder** as `DescriptionLookup/src` and the **Name** as `example.federal.us.ibm.com`, then click **Finish**.
5. Create or import the two Java source files shown in Listings 1 and 2 in the DescriptionLookup project: TenPrintSubmission.java and DescriptionLookup.java.

6. To generate JAX-WS web services from bottom up using the Java Bean artifacts, complete the following steps:
   1. Right-click DescriptionLookup.java and select Web Services => Create Web service.
   2. In the Service Deployment Configuration window, shown in Figure 6, specify WebSphere Business Monitor Server v7.0 on WebSphere Process Server for Server, and IBM WebSphere JAX-WS for Web service runtime. These selections enable you to deploy the generated JAX-WS web service to the Process Server V7 runtime.
3. In the Specify Service Project settings dialog, shown in Figure 6, specify DescriptionLookupService for the **Service project**, and DescriptionLookupServiceEAR for the **Service EAR Project**.
4. In the Client Environment Configuration window, specify WebSphere Business Monitor Server v7.0 on WebSphere Process Server for Server, and IBM WebSphere JAX-WS for Web service runtime, as shown in Figure 7.
5. In the Specify Client Projects dialog, shown in Figure 8, specify `DescriptionLookupServiceClient` for **Service project**, and `DescriptionLookupServiceClientEAR` for **Service EAR Project** to indicate the JAX-WS web service client project name, project enterprise archive project name and client project type of the dynamic web service.
7. Figure 9 shows a summary of the server and client generation settings for JAX-WS web service artifacts. At this time, we don't want to publish the service information to the UDDI registry or perform service monitoring using the TCP/IP monitor tool, so make sure that these options aren't checked. After reviewing the settings, click Next.
8. Next you need to choose the defaults for the JAX-WS web service delegate class name and the Java to WSDL mapping style. The JAX-WS runtime is capable of dynamically generating a deployment descriptor or WSDL file by inspecting annotations in the delegate class. However, we want to have these artifacts generated statically for our own review and reference. To accomplish this, check **Generate WSDL file into the project** and **Generate Web Service Deployment Descriptor**, as shown in Figure 10, then click **Next**.
9. On the next screen, shown in Figure 11, check **WSDL Target Namespace**, **Configure WSDL Service Name**, and **Configure WSDL Port Name**. These options will help you see the values being generated for reference and review purposes. Then click **Next**, and then **Next** again.
10. Next you need to generate a Java bean proxy, a JSP sample client that shows us how to call the Java bean proxy to invoke the web service implementation synchronously and asynchronously. We also want to have the wizard generate the deployment descriptor statically, as shown in Figure 12. To do this, check **Enable asynchronous invocation for generated client** and **Generate Web Service Deployment Descriptor**, and click **Next**.
Test the application

1. Next, we want to test the generated application using JAX-WS JSPs. We want the proxy to feature the specified methods and be deployed to the Process Server runtime, as shown in Figure 13. Specify JAX-WS JSPs for Test facility, then click Next, then Finish.
2. In the Servers view, verify that the JAX-WS DescriptionLookupService and DescriptionLookupServiceClient applications are deployed to the runtime and are running correctly, as shown in Figure 14.

**Figure 14. Verify application deployment**

(See a larger version of Figure 14.)

3. Click the `getDescription` link under **Methods** in the Web Services Test Client window, as shown in Figure 15.
Figure 15. Click getDescription in Web Services Test Client window

(See a larger version of Figure 15.)

4. We want to test the service implementation using a synchronous JAX-WS web service proxy client. To do this, specify Request for submissionType and CAR for transactionType, then click Invoke, as shown in Figure 16.

Figure 16. Specify synchronous request details

(See a larger version of Figure 16.)

5. Verify that the service response is as shown in the Result pane in Figure 17.
6. Next let’s test the service implementation using an asynchronous JAX-WS web service proxy client. To do this, repeat the test above, but specify **CAN** for `transactionType`, and check **Enable asynchronous invocation**. Verify that the service response is as shown in the **Result** pane in Figure 18.

**Figure 17. Verify synchronous service response**

(See a larger version of Figure 17.)

**Figure 18. Verify asynchronous service response**

(See a larger version of Figure 18.)

7. Verify that synchronous and asynchronous service calls are logging invocation confirmation messages to console output, as shown in Figure 19.
Create BPEL process model application to invoke the service

Now that you've created a service and service client, you're ready to compose a BPEL process model application to invoke this service. To accomplish this, you'll need to create a new business integration library and copy into this library your service artifacts, such as interface and implementation WSDLs and XML schema definition files.

1. Select **File => New => Library**.
2. In the Create a Library dialog, specify *ElectronicSubmissionLib* for **Library Name**, and click **Finish**, as shown in Figure 20.

   **Figure 20. Specify library name**

3. Select the **DescriptionLookupService_schema1.xsd** and **DescriptionLookupService.wsdl** files from the **Projects** pane of the Business Integration perspective, as shown in Figure 21, then select **Edit => Copy**.

   **Figure 19. Verify confirmation messages**

   (See a larger version of Figure 19.)
Figure 21. Select files to copy

4. Select the **ElectronicSubmissionLib** project from the **Projects** pane of the Business Integration perspective, as shown in Figure 22, then select **Edit => Paste**.
Figure 22. Paste files into project

5. To create a new interface to use to build the BPEL process model application, select **File => New => Interface**. Specify **ElectronicSubmissionLib** for **Module or library** and **DescriptionLookupProcess** for **Name**, then specify the **Namespace**, **Binding Style**, operation and parameter details as shown in Figure 23.
Figure 23. Specify details for DescriptionLookupProcess interface

6. Verify that two interface definitions are now listed under **ElectronicSubmissionLib => Interfaces** in the **Projects** pane of Business Integration perspective, as shown in Figure 24.

Figure 24. Verify interface creation

7. Now let’s create a simple BPEL process model application named **TenPrintSubmissionV0ProcessModule** to invoke the JAX-WS web service that you created and tested earlier. We’ll call this web services from a Java component implementation that we created and wired to BPEL process model application as an SCA module import, as shown in Figure 25.

Figure 25. BPEL application with SCA Java component import calling JAX-WS web service

8. Figure 26 shows the assembly diagram wiring to accomplish this integration. You can see that we have a process module wired to a downstream Java component as an SCA import.
9. We'll use the `DescriptionLookupProcess` interface we defined in `ElectronicSubmissionLib` as an interface to our BPEL process model. This requires you to define the BPEL process model application to be dependent upon `ElectronicSubmissionLib` module. Once you define this dependency, you'll be able to access library artifacts, as shown in Figure 27.
Figure 27. TenPrintSubmissionV0ProcessModule dependencies

Here is our simple BPEL process model application. The **Call Service** invoke step calls our Java component implementation, as shown on the **Properties** tab in Figure 28.

Figure 28. TenPrintSubmissionV0ProcessModule in process model editor

(See a larger version of Figure 28.)
Listing 3 shows the Java component implantation logic. This component logic is invoked by our BPEL process module through an SCA import wiring. The logic calls a downstream JAX-WS web service in four different invocation styles, as indicated by test cases 2 through 5. Test Case #1 invokes an inline Java bean implementation of the JAX-WS web service logic.

Listing 3. DescriptionLookupJavaImpl.java

```java
// import statements
...

public class DescriptionLookupJavaImpl {

    // generated methods
    ...

    public DataObject getDescription(DataObject aGetDescription) {

        String submissionType = aGetDescription.getDataObject("arg0").getString("submissionType");
        String txnType = aGetDescription.getDataObject("arg0").getString("transactionType");

        TenPrintSubmission tenPrintSubmission = new TenPrintSubmission();
        tenPrintSubmission.setSubmissionType(submissionType);
        tenPrintSubmission.setTransactionType(txnType);

        try {
            // # 1
            System.out.println("+++ Begin Test Case # 1 +++");
            System.out.println("+++ DescriptionLookupJavaImpl +++");
            tenPrintSubmission = new DescriptionLookup().getDescription(tenPrintSubmission);
            Thread.sleep(2000);

            // # 2
            System.out.println("+++ Begin Test Case # 2 +++");
            tenPrintSubmission = new SyncThinClient().invokeRemote(tenPrintSubmission);
            Thread.sleep(2000);

            // # 3
            System.out.println("+++ Begin Test Case # 3 +++");
            tenPrintSubmission = new PollingThinClient().invokeRemote(tenPrintSubmission);
            Thread.sleep(2000);

            // # 4
            System.out.println("+++ Begin Test Case # 4 +++");
            tenPrintSubmission = new CallbackThinClient().invokeRemote(tenPrintSubmission);
            Thread.sleep(2000);

            // # 5
            System.out.println("+++ Begin Test Case # 5 +++");
            tenPrintSubmission = new CallbackMEThinClient().invokeRemote(tenPrintSubmission);
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ...
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        ServiceManager sm = ServiceManager.INSTANCE;
        BOFactory bofactory = (BOFactory) sm.locateService("com/ibm/websphere/bo/BOFactory");
    }
```
DataObject aGetDescriptionResponse = bofactory.create("http://com.ibm.us.federal.example/", "getDescriptionResponse");
DataObject tenPrintSubmissionBO = bofactory.create("http://com.ibm.us.federal.example/", "tenPrintSubmission");
tenPrintSubmissionBO.setString("submissionType", submissionType);
tenPrintSubmissionBO.setString("transactionType", txnType);
tenPrintSubmissionBO.setString("transactionDescription", tenPrintSubmission.getTransactionDescription());
aGetDescriptionResponse.setDataObject("return", tenPrintSubmissionBO);
return aGetDescriptionResponse;
}

Test Case #2 is a synchronous thin client invocation of a JAX-WS web service hosted in a WebSphere Process Server V7 environment, as shown in Figure 29.

**Figure 29. BPEL process module with SCA synchronous Java component import calling JAX-WS web service**

Thin client implementation can be run as an unmanaged client (meaning the logic does not need to be hosted in a client or server container that is being managed by WebSphere middleware) or as a standalone client requiring only a compatible IBM JDK and a thin client JAR file, as shown in Figure 30. Both unmanaged and standalone client applications implement a Java main method with a thread of control.

**Figure 30. JAX-WS web service thin client support**

To invoke the JAX-WS web service method, you need to instantiate a proxy instance and invoke the method on the proxy synchronously, as shown in Listing 4.
Listing 4. SyncThinClient.java

```java
import example.federal.us.ibm.com.DescriptionLookupPortProxy;

public class SyncThinClient {

    public TenPrintSubmission invokeRemote(TenPrintSubmission tps) {
        try {
            DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
            tps = proxy.getDescription(tps);
            System.out.println("+++ SyncThinClient test +++");
            System.out.println("+++ Submission Type = " + tps.getSubmissionType() + " +++");
            System.out.println("+++ Transaction Description = " + tps.getTransactionDescription() + " +++");
            System.out.println("+++ Transaction Type = " + tps.getTransactionType() + " +++");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return tps;
    }
}
```

Test Case #3 is an asynchronous polling client invocation of a JAX-WS web service hosted in a WebSphere Process Server V7 environment, as shown in Figure 31. In this approach, a client sends a request to a web service endpoint in a non-blocking mode. The client gets the control of execution immediately after calling the service endpoint. The client does not wait for the response from the service endpoint, but instead receives a response object that it polls periodically to determine whether the service has responded. The client retrieves the actual response when the service responds and client polling succeeds, as shown in Figure 31. This approach of the client to invoking a web service makes the client more responsive and dynamic.

Figure 31. BPEL process module with SCA polling Java component import calling JAX-WS web service

To invoke the JAX-WS web service method, you'll need to complete the following steps, as shown in Listing 5:

1. Define the response variable and initialize it:
   ```java
   GetDescriptionResponse popDescResp = null;
   ```
2. Instantiate a proxy instance:
   ```java
   DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
   ```
3. Invoke the method on the proxy asynchronously:
   ```java
   Response<GetDescriptionResponse> resp = proxy.getDescriptionAsync(tps);
   ```
4. Check if response is ready to be retrieved:
   ```java
   while (!resp.isDone())
   ```
5. Retrieve response when available:
   popDescResp = resp.get();

**Listing 5. PollingThinClient**

```java
import java.util.Scanner;
import java.util.concurrent.ExecutionException;
import example.federal.us.ibm.com.DescriptionLookupPortProxy;
import example.federal.us.ibm.com.GetDescriptionResponse;
import javax.xml.ws.Response;

public class PollingThinClient {

    public TenPrintSubmission invokeRemote(TenPrintSubmission tps) {
        GetDescriptionResponse popDescResp = null;
        try {
            DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
            Response<GetDescriptionResponse> resp = proxy.getDescriptionAsync(tps);

            // Poll for the response.
            while (!resp.isDone()) {
                System.out.println("getDescription async still not complete.");
                Thread.sleep(200);
            }
            popDescResp = resp.get();
            System.out.println("getDescription async invocation complete.");

            System.out.println("+++ PollingThinClient test +++");
            System.out.println("+++ Submission Type = " +
                    popDescResp.getReturn().getSubmissionType() + " +++");
            System.out.println("+++ Transaction Description = " +
                    popDescResp.getReturn().getTransactionDescription() + " +++");
            System.out.println("+++ Transaction Type = " +
                    popDescResp.getReturn().getTransactionType() + " +++");
        } catch (InterruptedException e) {
            System.out.println(e.getCause());
        } catch (ExecutionException e) {
            System.out.println(e.getCause());
        }

        return popDescResp.getReturn();
    }
}
```

Test Case #4 is an asynchronous callback client invocation of a JAX-WS web service with asynchronous behavior at programming model by client when invoking a web service hosted in a WebSphere Process Server V7 environment, as shown in Figure 32. In this invocation model, the client implements the `javax.xml.ws.AsynchHandler` interface to accept and process the response object received from service. The client-provided `AsynchHandler` callback handler code is run when an asynchronous response is received from the service. The client receives the response object in the form of a `javax.xml.ws.Response` object. The client performs a get operation on the response
object to retrieve the response content. This approach of the client invoking a web service makes the client more responsive and dynamic.

**Figure 32. BPEL process module with SCA callback Java component import calling JAX-WS web service**

![Diagram of BPEL process module with SCA callback Java component import calling JAX-WS web service](image)

To invoke the JAX-WS web service method, you need to complete the following steps, as shown in Listing 6 and 7):

1. Instantiate a proxy instance
   ```java
   DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
   ```
2. Instantiate a callback handler instance
   ```java
   CallbackThinClientHandler callbackHandler = new CallbackThinClientHandler();
   ```
3. Invoke the method on the proxy asynchronously with callback handler
   ```java
   Future<?> response = proxy.getDescriptionAsync(tps, callbackHandler);
   ```
4. Retrieve response in callback handler
   ```java
   popDescResp = resp.get();
   ```

### Listing 6. CallbackThinClient

```java
import java.util.Scanner;
import java.util.concurrent.Future;

public class CallbackThinClient {

    public TenPrintSubmission invokeRemote(TenPrintSubmission tps) {
        DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
        CallbackThinClientHandler callbackHandler =
            new CallbackThinClientHandler();
        Future<?> response = proxy.getDescriptionAsync(tps, callbackHandler);
        System.out.println("+++ Wait 5 seconds +++");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("+++ CallbackThinClient test async end +++");
        return callbackHandler.getResponse();
    }
}
```

### Listing 7. CallbackThinClientHandler

```java
import java.util.Scanner;
import java.util.concurrent.Future;
```
public class CallbackThinClient {

    public TenPrintSubmission invokeRemote(TenPrintSubmission tps) {
        DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
        CallbackThinClientHandler callbackHandler =
            new CallbackThinClientHandler();
        Future<?> response = proxy.getDescriptionAsync(tps,
            callbackHandler);
        System.out.println("+++ Wait 5 seconds +++");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("+++ CallbackThinClient test async end +++");
        return callbackHandler.getResponse();
    }
}

Test Case #5 is an synchronous callback client invocation of a JAX-WS web service with asynchronous behavior at the programming model, and pattern of message exchanges on the wire by the client when invoking a web service hosted in a WebSphere Process Server V7 environment, as shown in Figure 33. In this invocation model, the client uses WS-Addressing to indicate to the service the ReplyTo endpoint reference (EPR) value to be used by the service when responding to the client’s request. The client then listens on a separate HTTP channel to receive the response messages sent by the service. The service initiates a separate HTTP channel on the ReplyTo EPR value provided by the client to send the response back to the client. This is an IBM extension of the JAX-WS specification to provide asynchronous message exchange support on the wire. To use an asynchronous message exchange pattern on the wire, you must set the custom property com.ibm.websphere.webservices.use.async.mep to true on the client request context. This approach of the client invoking a web service makes the client more responsive and dynamic.

Figure 33. BPEL process module with SCA callback ME Java component import calling JAX-WS web service

To invoke JAX-WS Web Service method, you need to complete the following steps, as shown in Listing 8:

1. Instantiate a proxy instance
   ```java
   DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
   ```
2. Instantiate binding provider object so that client context can be obtained and set
   ```java
   BindingProvider bp = (BindingProvider) proxy._getDescriptor().getProxy();
   ```
3. Set the client context
   ```java
   bp.getRequestContext().put ("com.ibm.websphere.webservices.use.async.mep",
   Boolean.TRUE);
   ```
4. Instantiate a callback handler instance
   ```java
   CallbackThinClientHandler callbackHandler = new CallbackThinClientHandler()
   ```
5. Invoke the method on the proxy asynchronously with callback handler
   
   ```java
   Future<?> response = proxy.getDescriptionAsync(tps, callbackHandler);
   ```

6. Retrieve response in callback handler

   ```java
   popDescResp = resp.get();
   ```

Listing 8. CallbackMEThinClient.java

```java
package example.federal.us.ibm.com;

import java.util.Scanner;
import java.util.concurrent.Future;
import javax.xml.ws.BindingProvider;

public class CallbackMEThinClient {

    public TenPrintSubmission invokeRemote(TenPrintSubmission tps) {
        DescriptionLookupPortProxy proxy = new DescriptionLookupPortProxy();
        BindingProvider bp = (BindingProvider) proxy._getDescriptor().getProxy();
        bp.getRequestContext().put("com.ibm.websphere.webservices.use.async.mep", Boolean.TRUE);
        CallbackThinClientHandler callbackHandler = new CallbackThinClientHandler();
        Future<?> response = proxy.getDescriptionAsync(tps, callbackHandler);
        System.out.println("+++ Wait 5 seconds +++");
        try {
            Thread.sleep(5000);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("+++ CallbackThinMEClient test async end +++");
        return callbackHandler.getResponse();
    }
}
```

Figure 34 shows the **Server Logs** view output when all the use cases are executed.

**Figure 34. TenPrintSubmissionV0ProcessModule server log output**

(See a larger version of Figure 34.)

Figure 35 shows the **Console** view output when all the use cases are executed.
In the following use case, shown in Figure 36, the BPEL process module calls a passthru mediation module using an SCA import wiring. The pass-thru mediation module in turn calls the JAX-WS web service. In this usage pattern, you can easily perform mediation functions including auditing, tracing, validations, transformations and others, using WebSphere Enterprise Service Bus’s pre-engineered mediation flow primitives.

Figure 37 shows the assembly diagram wiring for this use case.

Figure 38 shows the mediation flow editor’s solution composition for the request and response flows.
Figure 39 shows the deployed applications set consisting of the BPEL process module, the mediation module, and the JAX-WS web service applications (client and server).

**Figure 39. TenPrintSubmissionV1ProcessModule deployed applications**

![Image](image_url)

(See a larger version of Figure 39.)

Figure 40 shows the testing of the solution logic with the associated service response.

**Figure 40. TenPrintSubmissionV1ProcessModule testing**

![Image](image_url)

(See a larger version of Figure 40.)

In the following use case, shown in Figure 41, the BPEL process module calls a mediation module with a **Service Invoke** primitive, using an SCA import wiring. The mediation module in turn calls the JAX-WS web service. In this usage pattern, you can invoke JAX-WS web service functions, either synchronously or asynchronously, with configurable multiple retries and configurable delays between multiple retries. Additionally, you can easily perform mediation functions consisting of auditing, tracing, validations, transformations and others, using WebSphere Enterprise Service Bus's pre-engineered mediation flow primitives.
Figure 41. BPEL process module with SCA Service Invoke primitive mediation module import calling JAX-WS web service

Figure 42 shows the assembly diagram wiring for this use case.

**Figure 42. TenPrintSubmissionV2ProcessModule assembly diagram**

(See a larger version of Figure 42.)

Figure 43 shows the mediation flow editor's solution composition for request flow.

**Figure 43. TenPrintSubmissionV2ProcessModule mediation module**

(See a larger version of Figure 43.)

Figure 44 shows the deployed applications set consisting of the BPEL process module, the mediation module, and the JAX-WS web service applications (client and server).

**Figure 44. TenPrintSubmissionV2ProcessModule deployed applications**

(See a larger version of Figure 44.)

Figure 45 shows the testing of the solution logic with the associated service response.
In the following use case, shown in Figure 46, the BPEL process module calls a custom primitive mediation module using an SCA import wiring. The mediation module in turn calls the JAX-WS web service. In this usage pattern, you can easily perform custom mediation functions consisting of metering, custom validations and others, using WebSphere Enterprise Service Bus's pre-engineered mediation flow primitives.

Figure 46. BPEL process module with SCA custom mediation primitive mediation module import calling JAX-WS web service

Figure 47 shows the assembly diagram wiring for this use case.

Figure 47. TenPrintSubmissionV3ProcessModule assembly diagram

(See a larger version of Figure 47.)

Figure 48 shows the mediation flow editor's solution composition for request and response flows.
Figure 48. TenPrintSubmissionV3ProcessModule mediation module

(See a larger version of Figure 48.)

Figure 49 shows the deployed applications set consisting of the BPEL process module, mediation module, and the JAX-WS web service applications (client and server).

Figure 49. TenPrintSubmissionV3ProcessModule deployed applications

(See a larger version of Figure 49.)

Figure 50 shows the testing of the solution logic with the associated service response.

Figure 50. TenPrintSubmissionV3ProcessModule testing

(See a larger version of Figure 50.)

Summary

Congratulations! You've successfully designed, developed and tested an integration solution featuring business process, mediation, and JAX-WS web service components. You've successfully used WebSphere Integration Developer V7 to design, build and test a bottom-up JAX-WS web service from both client and server-side perspectives. You've composed BPEL process module.
applications to invoke the JAX-WS web service using a Java component implementation and a mediation module application. You've explored the composition of mediation module applications with pass-thru or Service Invoke or Customer mediation module primitives. You've also explored various options that are available to from the client side for invoking the service using synchronous or asynchronous polling or asynchronous callback or asynchronous callback with message exchange styles with the associated design implications.

Acknowledgments

The author would like to gratefully acknowledge help received from Sujatha Perepa, IT Architect, IBM Federal Software Group, Washington, D.C., in reviewing the article and suggesting helpful improvements.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Interchange file with sample projects</td>
<td>DescriptionLookupService.zip</td>
<td>234KB</td>
</tr>
</tbody>
</table>
Related topics

- WebSphere Integration Developer Information Center
- Rational Application Developer V7.5 Programming Guide
- List of mediation primitives
- Service Invoke mediation primitive
- Custom mediation primitive
- The Integrated Automated Fingerprint Identification System (IAFIS)
- Electronic Fingerprint Transmission Specification
- developerWorks BPM zone: Get the latest technical resources on IBM BPM solutions, including downloads, demos, articles, tutorials, events, webcasts, and more.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)