Java CAPS 6 and OpenESB

Scheduler for JCA and JBI projects

Michael Czapski, July 2008

Table of Contents

1 Introduction

Java CAPS 5.x used to have a Scheduler eWay. Java CAPS 6 also has the Scheduler eWay but only on the Repository-based side. At this point in time there is no Scheduler JCA Adapter or a Scheduler Binding Component. Why would one be bothered by that? One would be bothered because there are business requirements that call for scheduling of activities. The one that comes to mind immediately is polling an FTP server for a file which to transfer. For polling the local file system there is the Batch Inbound JCA, which was used in solutions discussed in JCA Notes 2 and 3. For Batch FTP JCA there is no such thing.

Rather then ignoring the issue of lack of the Scheduler JCA Adapter I determined to see what can be done to provide this functionality for non-Repository-based Java CAPS 6 solutions.

When asked, one of my colleagues in the US suggested that EJB Timers are the way to go and provided the links to the material. I looked at what was discussed, threw up my hand in the air and exclaimed. I will not quote what I said. In short, EJB Timers may be all very well for a competent Java EE developer but not for a regular Integration, or SOA, developer. EJB Timers are, in my view, way too complex to implement and do not offer sufficient advantage over a Scheduler eWay to make it worth while to spend the time developing a solution that uses them.

The next thing I looked at was the open source Quarz scheduler, which also turned up to require more effort then I considered worth while for the Notes.

I felt that the simplest thing to do will be to use an external scheduler, a native one, provided by the OS. For Windows, on which I develop for the Notes, there is the “Scheduled Tasks” scheduler. For Unix there are Cron facilities. Both are well know and typically good enough in terms of timer resolution and scheduling flexibility. Above all else, using one does not require me to write scheduler code myself, merely write the code that triggers my solution when the scheduled event fires.

So, this Note walks through implementation of a Scheduler solution, which can be used to trigger a Batch FTP JCA solution or any other JCA-based or JBI-based solution that has to be triggered to some schedule.

2 Design

After a number of attempts, starting with a stand-alone JMS sender, through a TCP/IP-based sender/listener, I decided that a simple web service client/server

solution will be the fastest and most straight forward to develop in NetBeans 6.1, the Java CAPS 6 and OpenESB IDE.

The idea is that a native OS scheduler can be scheduled to execute an application it is configured to execute, and that the application can be configured to read its command line parameters and use them to either modify its behavior or to do something else with them, like pass them to a partner application.

To accomplish what needs be accomplished we need a stand-alone Java Application client and a server hosted in the Application Server.
The Java Application will have a command line of the form:

```
java –jar clientApp.jar server-host server-port “rest of command line”
```

It will parse its command line parameters, use the server-host and server-port to dynamically configure the end point address it will use to connect to its server counterpart, and will pass the “rest of command line” to it as a string. Any unquoted words will be concatenated, using a pipe character “|” delimiter, into a single string. A quoted string will be treated like a word. So, if the “rest of command line” looked like:

```
This is “a test of” command “line parameters”
```

Then the concatenated string will be:

```
|This|is|a test of|command|line parameters|
```

The server implementation will not do anything to the sting it receives. It will be up to the component that gets triggered to parse the string and do what it will on the basis of information it contains.

The client and the server will use SOAP over HTTP to communicate. Whilst this is possibly the most inefficient way of inter-process communication it has the advantage of being best supported by NetBeans tooling. It is faster to implement then the more efficient alternatives.

The client will retrieve its command line parameters, dynamically modify the end point address of the service, concatenate the rest of the command line and invoke the service passing it the concatenated string. The service will be a One-Way Operation service so the service will return no response. Once the service is invoked and the message is passed the client application will terminate.

The server implementation will receive the message and immediately queue it to a JMS Queue, which it will be configured to use. This is so that trigger messages can be queued and processed or ignored as the application to be triggered sees fit. It is also so that the design of the triggered application is simple. It needs to be a JMS receiver, which is easy enough to do both using the JCA and the JBI facilities.

3 Creating the Project Group

As with all Notes so far, the first thing to do is to create a NetBeans Project Group, Scheduler_PG, in the appropriate directory, to contain all development artifacts associated with this Note. Figures 3-1 and 3-2 illustrate key steps.

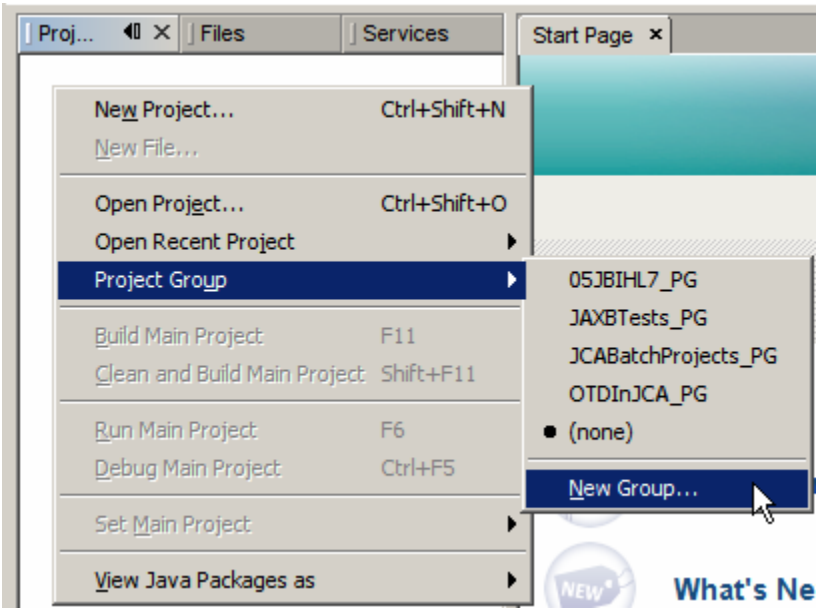

Figure 3-1 Starting New Group Wizard

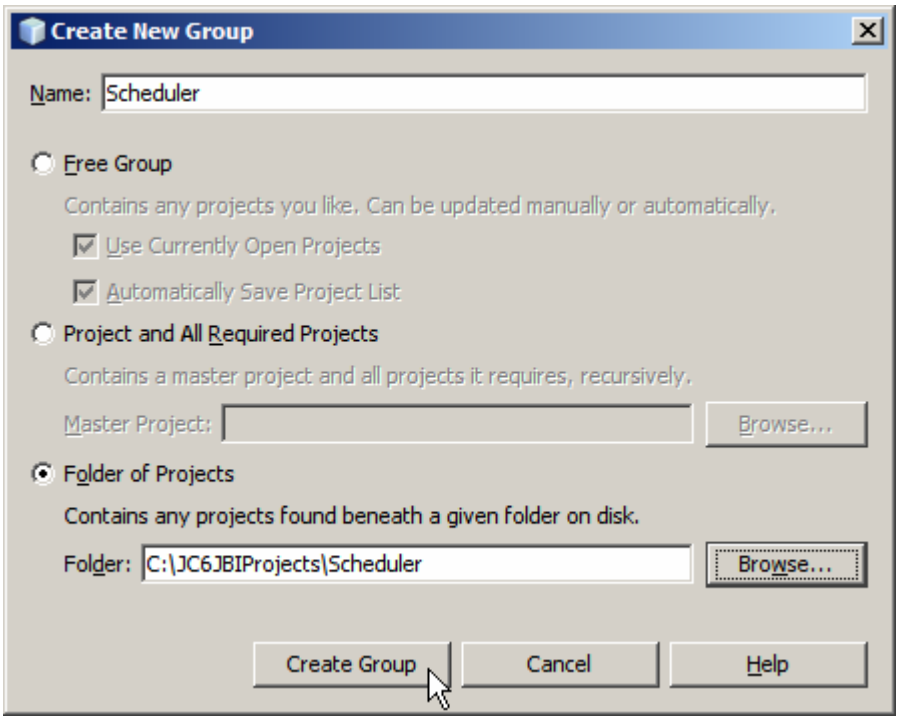

Figure 3-2 Naming the Project Group and setting directory location

4 Creating the WSDL

Since the client and the server will communicate using SOAP over HTTP it is desirable to define the interface to which both must conform. We will create a WSDL Document which defines that interface in a project of its own. Let's create a New ->

Enterprise -> EJB Module, Scheduler_WSDL, to contain the WSDL. Figures 4-1 and 4-2 illustrate key steps.

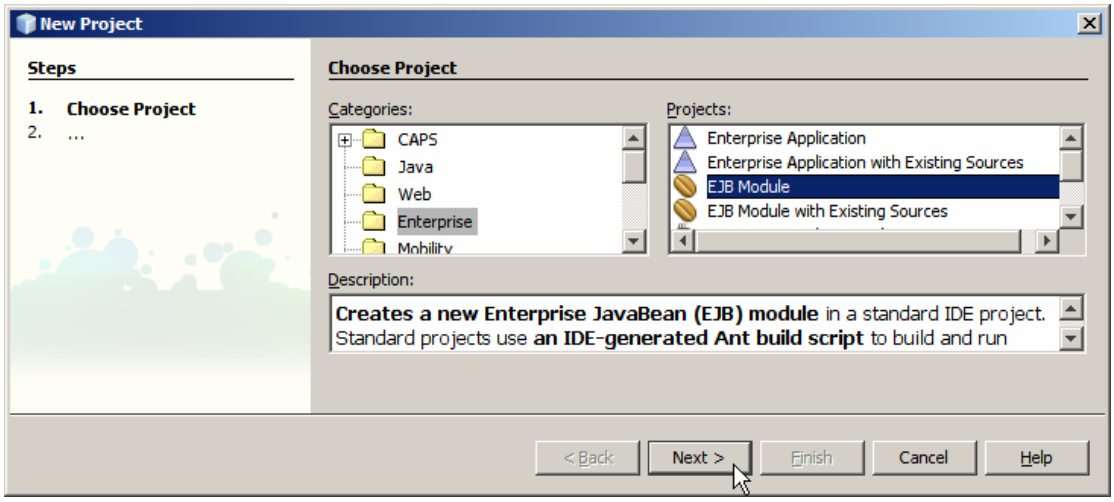

Figure 4-1 Choosing project type

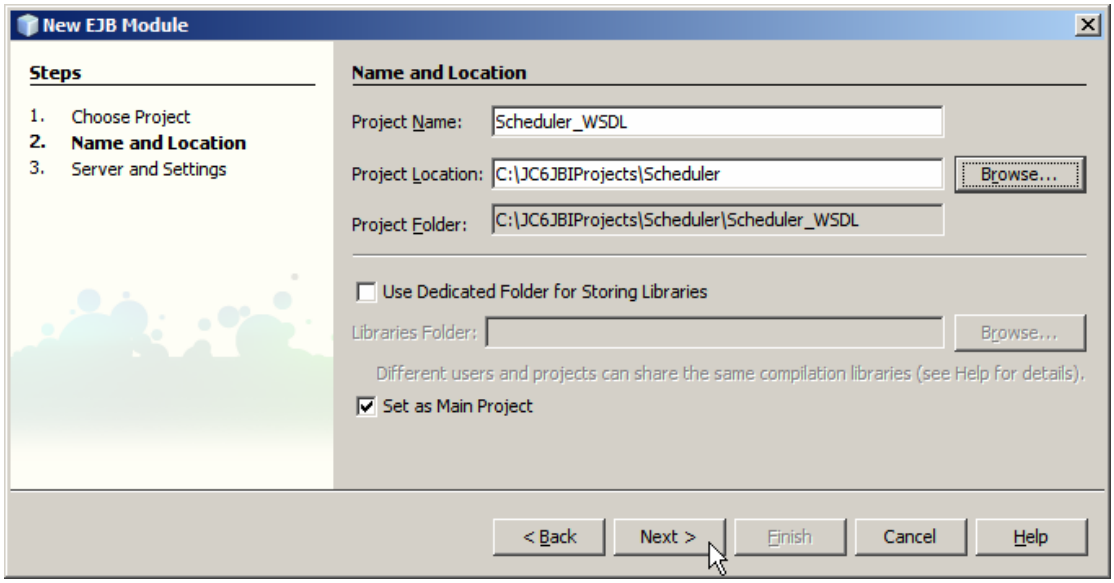

Figure 4-2 Naming the project and setting its location

Once the project is created, let's create a WSDL, named "Scheduler", with a One-Way operation, opSubmitTrigger, a single part xsd:string message, sTrigger, that uses SOAP binding. Figures 4-3 through 4-6 illustrate the steps.

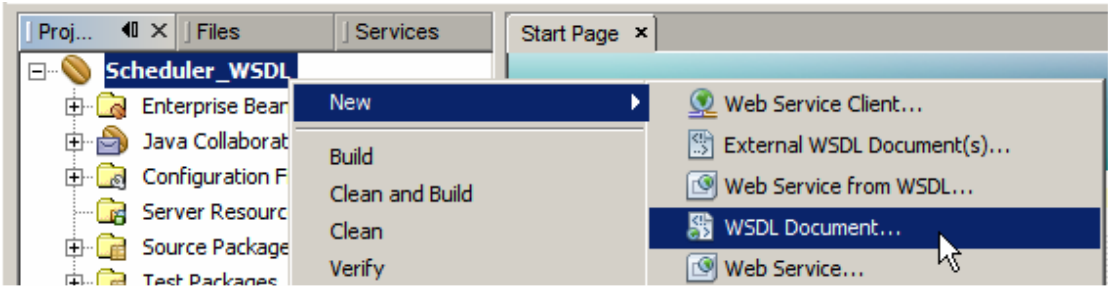

Figure 4-3 Start New WSDL Document Wizard

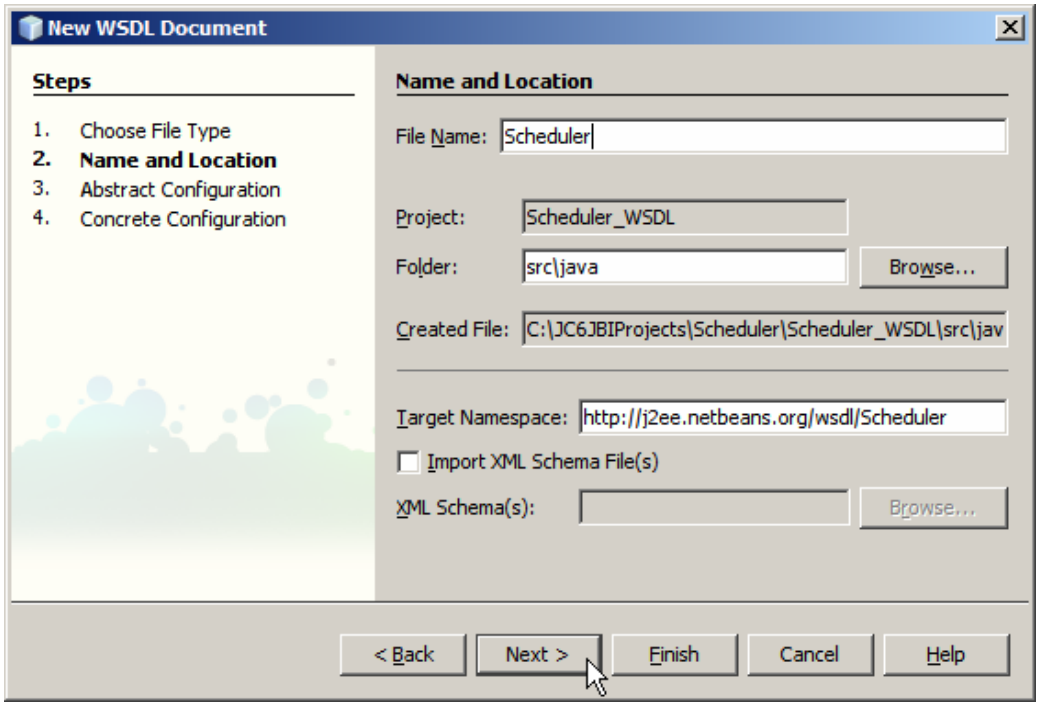

Figure 4-4 Name the WSDL file

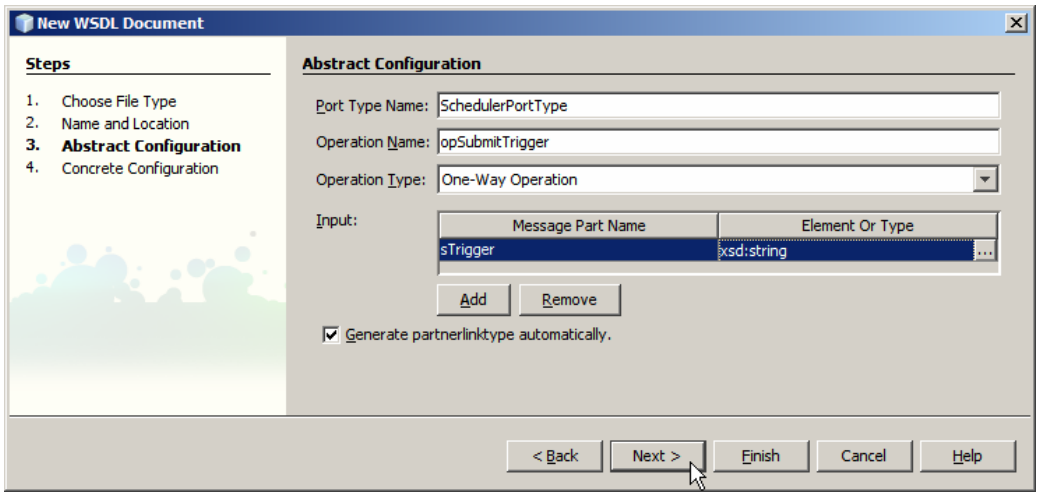

Figure 4-5 Naming the operation, choosing the type and naming the message part

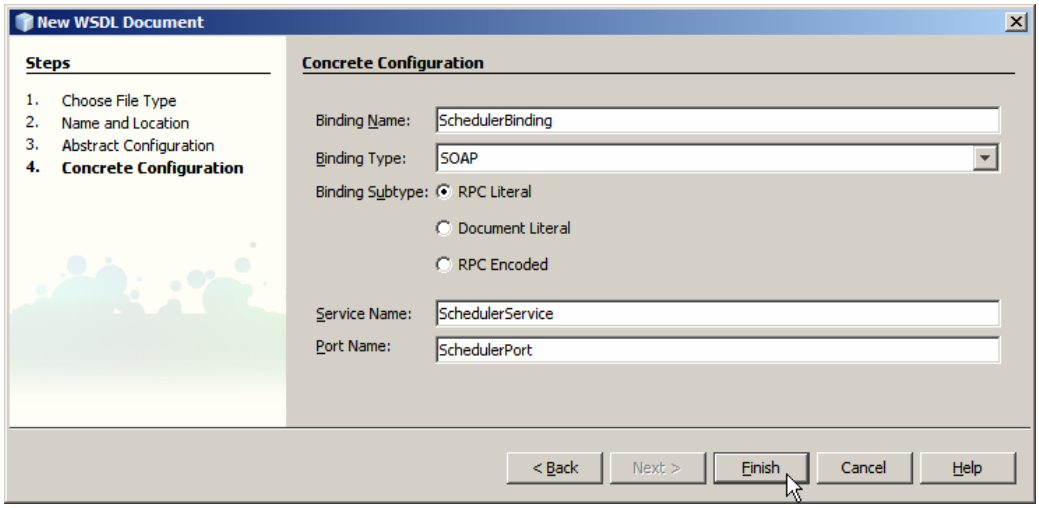

Figure 4-6 Choosing SOAP binding and completing the wizard

Once the WSDL is created, locate the soap:address node and change the location property to use explicit port number. Use your application server's default HTTP Listener port number. For a default installation it will be 8080. Figure 4-7 illustrates this.

Figure 4-7 Setting explicit port number in the soap:address location property

5 Creating a Stand-alone Client

Now we need to create a Java Application. Since this is not a Java programming tutorial I will not dwell on the code. A Java programmer will likely have written it differently. It suffices for our need.

Let's create a New Project -> Java -> Java Application, as illustrated in Figures 5-1 and 5-2. Let's make sure that the name of the main class is identical to the WSDL port name. In our WSDL port name is SchedulerPort so the name of the main class will also be SchedulerPort. For reasons unknown Web Service Reference, which we will produce later, modifies the implemented end point servlet context string so that it uses the class name instead of the port name specified in the WSDL. By making sure the main class name is the same as the port name we avoid issues arising from the end point servlet context becoming different from that used in the WSDL.

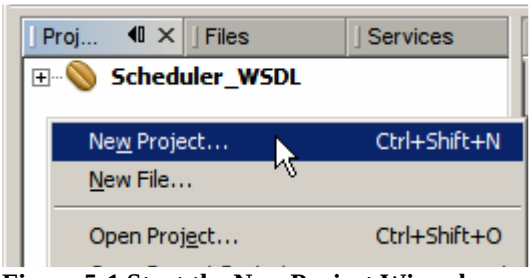

Figure 5-1 Start the New Project Wizard

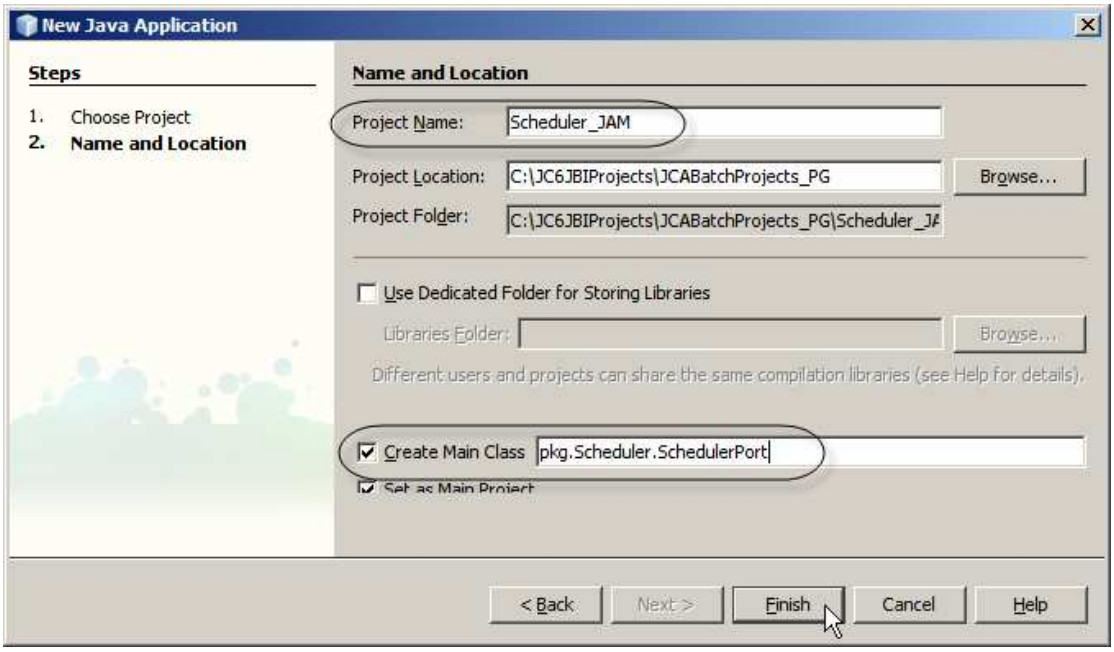

Figure 5-2 Naming the project, the package and the main class

Let's add the a couple of imports following the package statement and before the public class statement. Listing 5-1 shows the source code of the complete class.

```
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.ws.BindingProvider;
```

Let's add the following code to the main method. This code concerns itself with parsing the command line arguments.

```
String sHost = null;
int iPort = 0;
String sExtraInfo = "";

if (args.length < 1) {
    showUsage();
    return;
}

sHost = args[0];
System.out.println("Host:\t" + sHost);

if (args.length < 2) {
    System.out.println("Error:\tPort Number expected\n");
    showUsage();
    return;
}

try {
    iPort = Integer.parseInt(args[1]);
    System.out.println("Port:\t" + iPort);

} catch (Exception e) {
    System.out.println
      ("Error:\tPort must be an integer - have \""
      + args[1] + "\"\n");
```

```
    showUsage();
    return;
}

for (int i = 2; i < args.length; i++) {
    sExtraInfo += args[i] + "|";
}
if (sExtraInfo != null) {
    System.out.println("Extra:\t" + sExtraInfo);
}

// ------------------------------------------------------------
```

Let's save what we have so far and create a web service reference to use in the code.

Let's create a New -> Web Service Client, locate the Scheduler.wsdl in the Scheduler_WSDL project's source directory, and Finish. Figures 5-3 and 5-4 illustrate the steps.

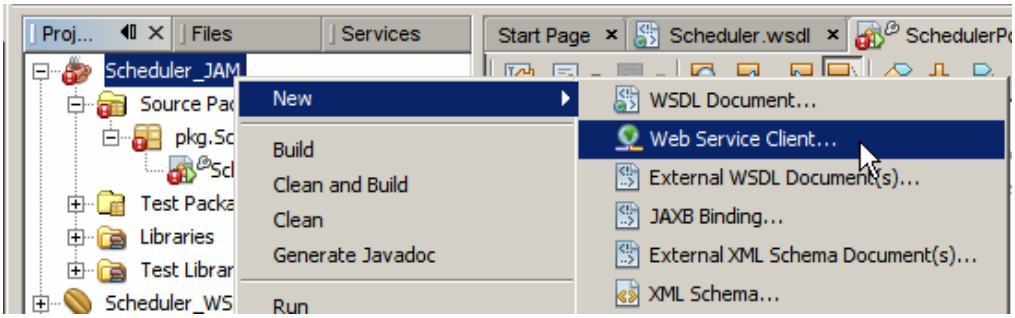

Figure 5-3 Start New Web Service Client wizard

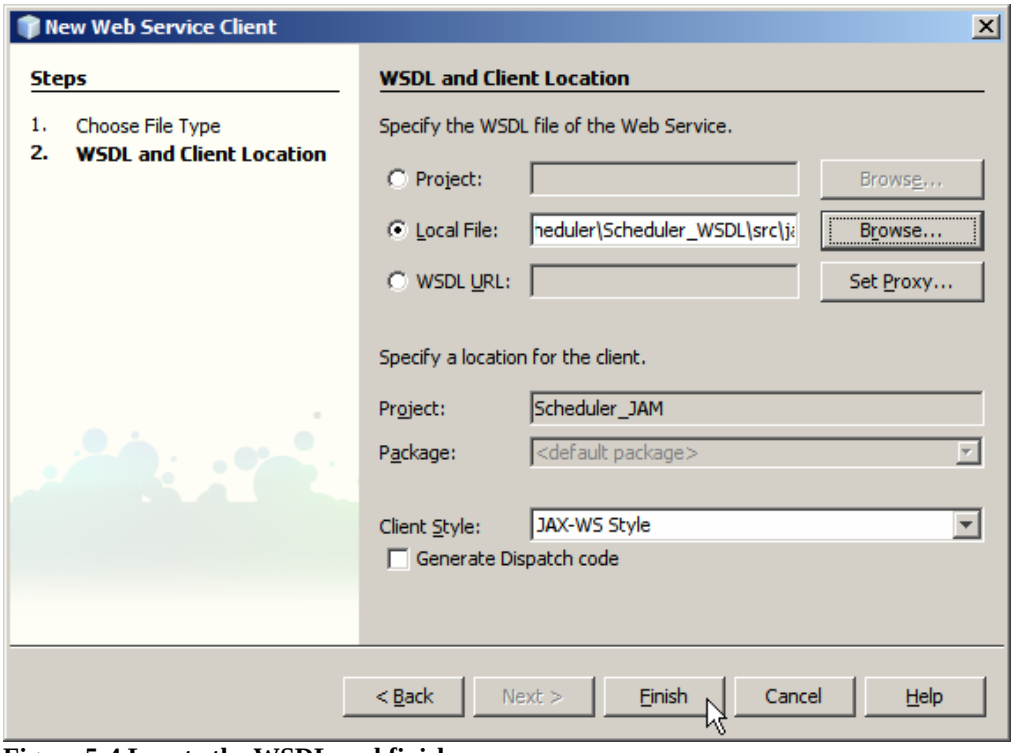

Figure 5-4 Locate the WSDL and finish

On completion of the Wizard we have a Web Service Reference folder, mush like the one in Figure 5-5.

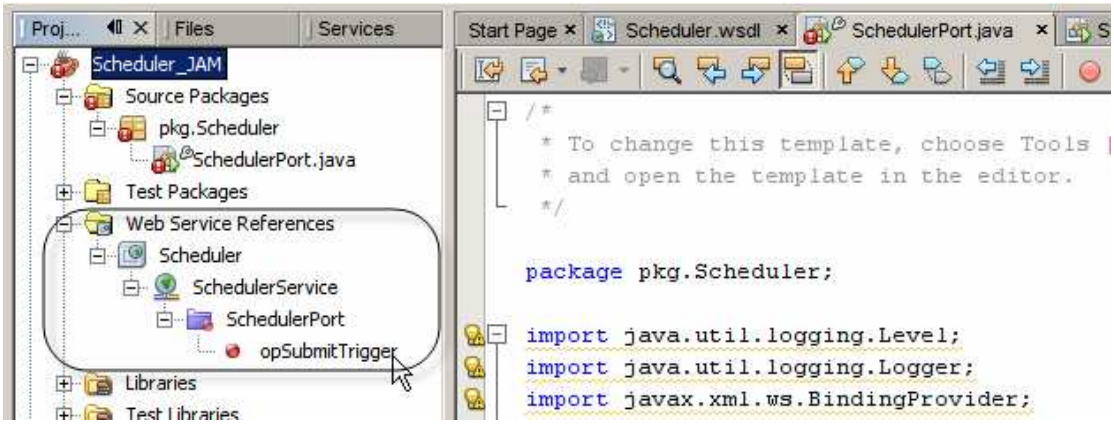

Figure 5-5 Web Service Reference

Let's expand the node tree under the web service reference node and drag the opSubmitTrigger operation to our SchdulerPort.java source window following the "lots of dashes" comment, as illustrated in Figure 5-6.

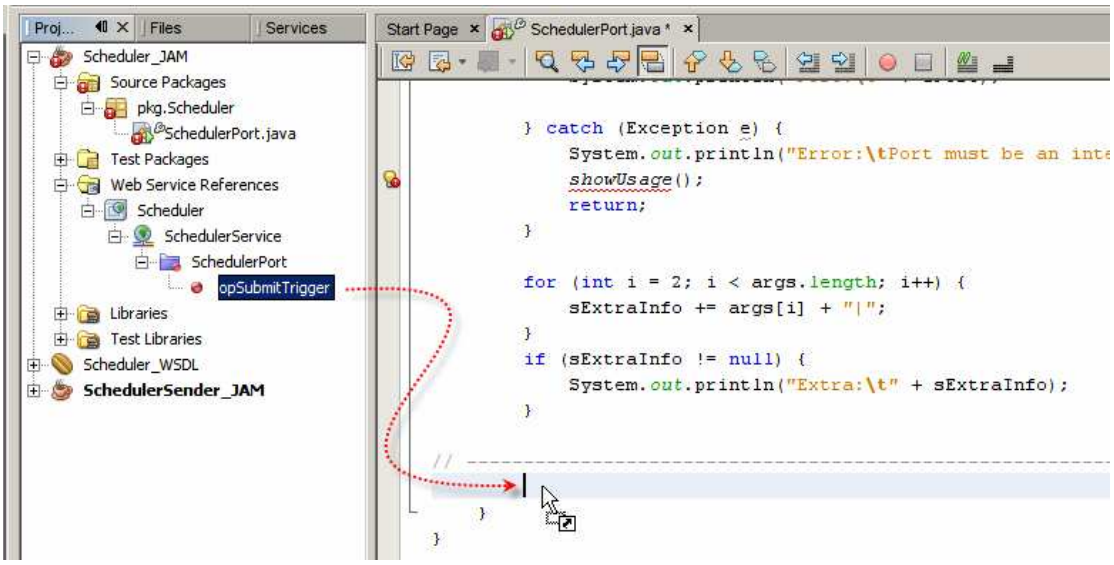

Figure 5-6 Drag the web service operation to the source code window

This adds a rather ugly slab of Java code that invokes the web service. Let's re-organise the code somewhat so it is more readable.

Let's copy the text highlighted in Figure 5-7 and make an import statement out of it, as shown in Figure 5-8.

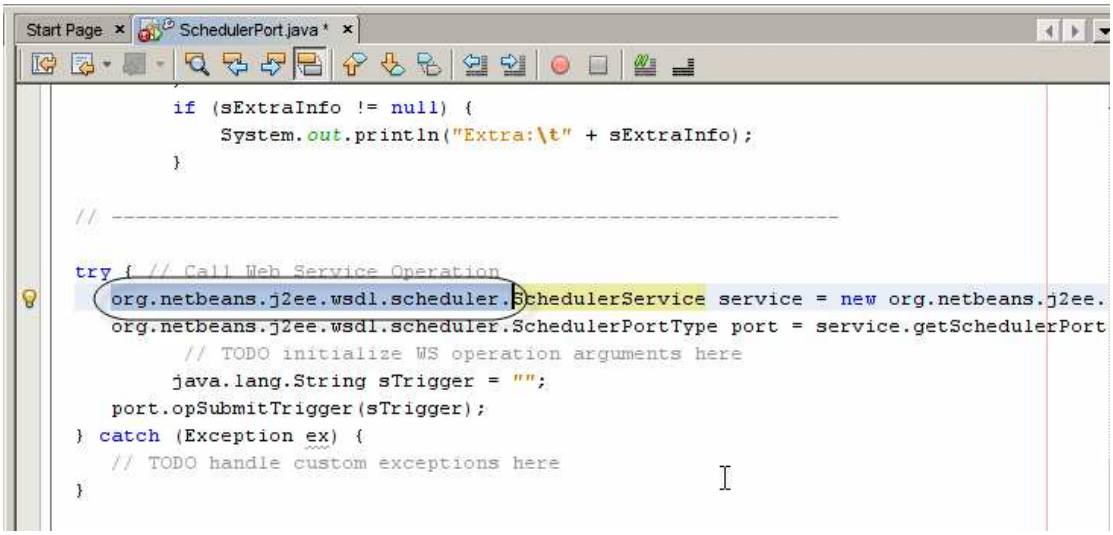

Figure 5-7 Copy org.netbeans.j2ee.wsdl.scheduler

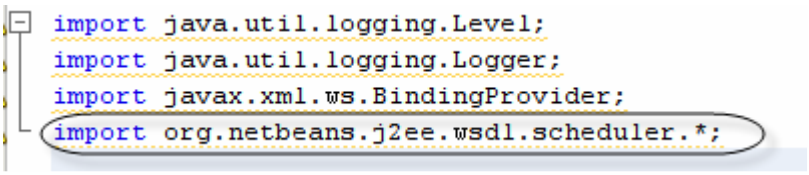

Figure 5-8 Add an import statement

Now reformat the code as shown in Figure 5-9.

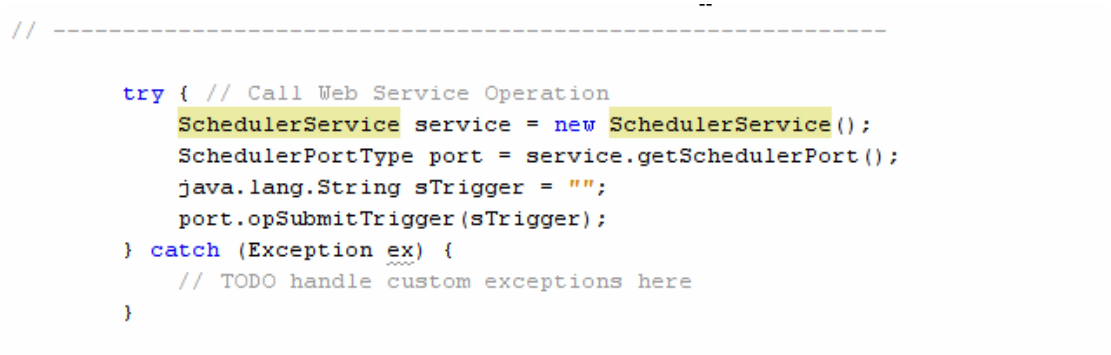

Figure 5-9 Reformat the code

With the code as shown, the end point address is fixed. We need to make it dynamic to the point where the host address and port number can be changed at runtime. Modify the code so it reads as shown in Figure 5-10.

Figure 5-10 Add or modify code

The complete source is shown in Listing 5-1.

```
package pkg.Scheduler;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.ws.BindingProvider;
import org.netbeans.j2ee.wsdl.scheduler.*;

/**
 *
 * @author mczapski
 */
public class SchedulerPort {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String sHost = null;
        int iPort = 0;
        String sExtraInfo = "";

        if (args.length < 1) {
            showUsage();
            return;
        }

        sHost = args[0];
        System.out.println("Host:\t" + sHost);

        if (args.length < 2) {
            System.out.println("Error:\tPort Number expected\n");
            showUsage();
            return;
        }
```

```
        try {
            iPort = Integer.parseInt(args[1]);
            System.out.println("Port:\t" + iPort);

        } catch (Exception e) {
            System.out.println("Error:\tPort must be an integer - have \""
                    + args[1] + "\"\n");
            showUsage();
            return;
        }

        for (int i = 2; i < args.length; i++) {
            sExtraInfo += args[i] + "|";
        }
        if (sExtraInfo != null) {
            System.out.println("Extra:\t" + sExtraInfo);
        }

// ---------------------------------------------------------

        try { // Call Web Service Operation
            SchedulerService service = new SchedulerService();
            SchedulerPortType port = service.getSchedulerPort();
            ((BindingProvider)port).getRequestContext().put
                    (BindingProvider.ENDPOINT_ADDRESS_PROPERTY
                    ,"http://" + sHost + ":" + iPort
                    + "/SchedulerService/SchedulerPort");
            port.opSubmitTrigger(sExtraInfo);
        } catch (Exception ex) {
            Logger.getLogger(SchedulerPort.class.getName()).log(Level.SEVERE
                    ,"Failed to send trigger as service request", ex);
            ex.printStackTrace();
        }
    }

    private static void showUsage() {
        System.out.println("Usage:\n\tjava SubmitScheduledTrigger "
                + "<service-host> "
                + "<service-port> \""
                + "<additional-trigger-information>\"\n");
    }

}
```

Listing 5-1 Client source code

Build the project, Scheduler_JAM. This will produce the Scheduler_JAM.jar file and a lib directory with all JAR files required to execute the client.

The command line to invoke the application will be similar to that shown below. The host name, port number and additional trigger information will vary, as will the JDK location and the current working directory.

```
C:\JC6JBIProjects\Scheduler\Scheduler_JAM\dist>c:\jdk1.6.0_02\bin\jav
a -Djava.endorsed.dirs=.\lib -jar Scheduler_JAM.jar localhost 58080
Hi there
```

Note that `–Djava.endorsed.dirs=.\lib` is required and must point at the directory containing all required JARs. This directory is constructed and populated by the build process.

To deploy the stand-alone client to an environment simply copy the Scheduler_JAM.jar and the entire lib directory to where you need it.

6 Creating a JCA Trigger Receiver

This implementation will only be possible in Java CAPS 6. To my knowledge OpenESB does not have a concept of JCA Adapters and none are available in the OpenESB distribution. If you need a solution that can be used in an OpenESB solution see next section – Creating a JBI Trigger Receiver.

In this section we will implement a web service that will accept the trigger string and deposit it in the hard-coded JMS Queue. The Queue name could be looked up in JNDI or could be provided as a component of the trigger message. This is left as an exercise for the reader. For this Note we will keep everything simple as much as possible.

Let's create a New Project -> Enterprise -> EJB Module, SchedulerListener_EJBM. Figures 6-1 through 6-3 illustrate key points.

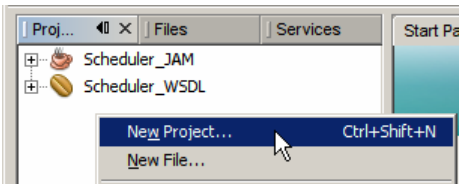

Figure 6-1 Start the New Project Wizard

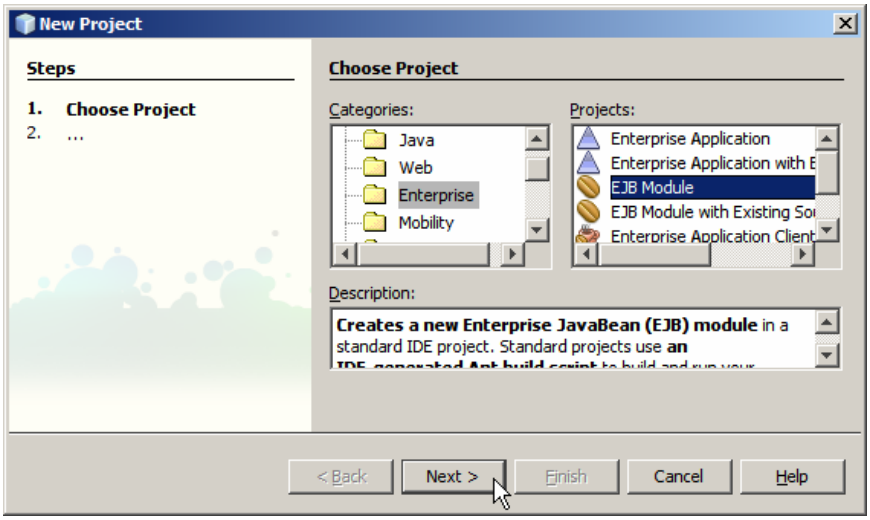

Figure 6-2 Enterprise -> EJB Module

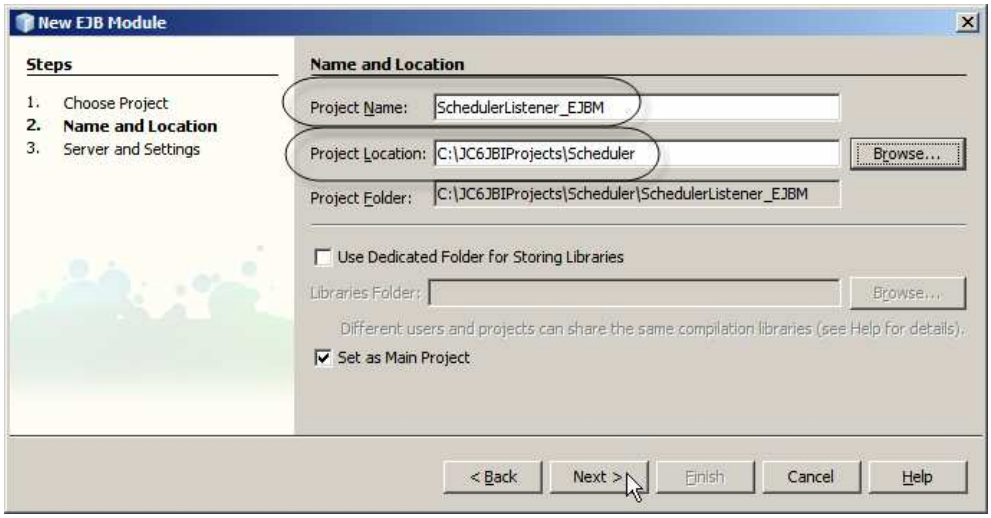

Figure 6-3 Give it the name and choose the location for project artefacts

Now that the project structure is constructed let's create a Web Service from WSDL, call it SchedulerPort and use the WSDL form the Scheduler_WSDL project, which we created at the beginning. Figures 6-4 and 6-5 illustrate key points. Here it is important to make sure that the implementation class name is the same as the WSDL Port name.

Figure 6-4 Start the New Web Service from WSDL Wizard

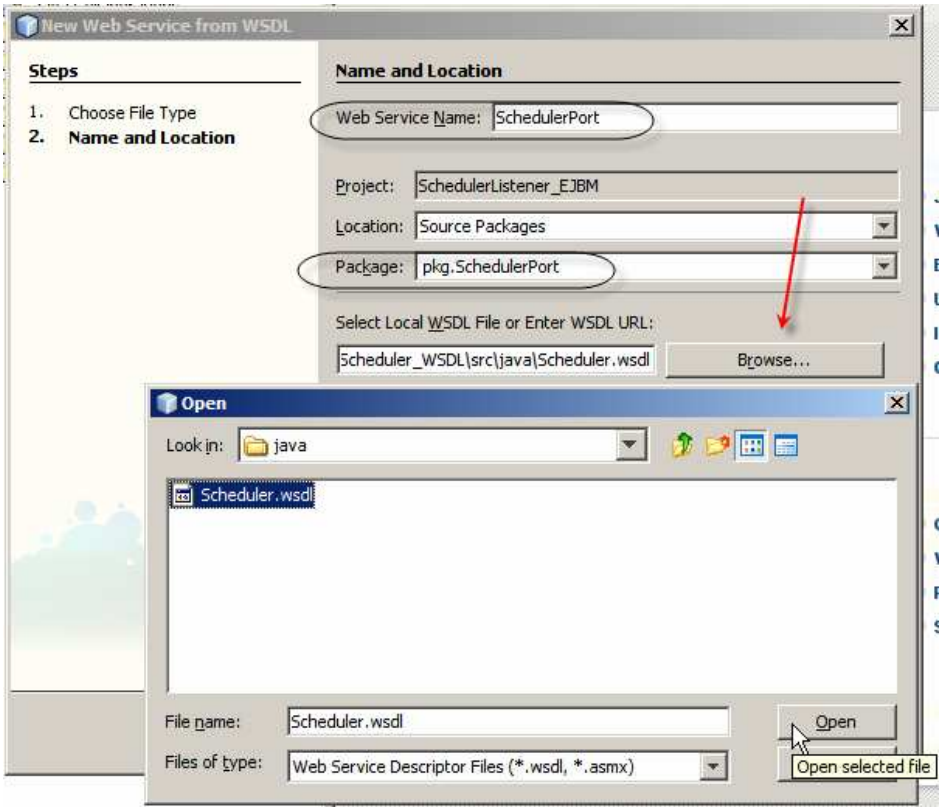

Figure 6-5 Name the service and the package, and choose the WSDL to implement

In due course the web service implementation skeleton is generated. Switch to the source code window and delete lines 21 and 22, see Figure 6-6, and replace them with a blank line.

Figure 6-6 Source code window – delete lines 21 and 22 and insert a blank line instead

From the Palette drag the JMSOTD onto the source window as shown in Figure 6-7.

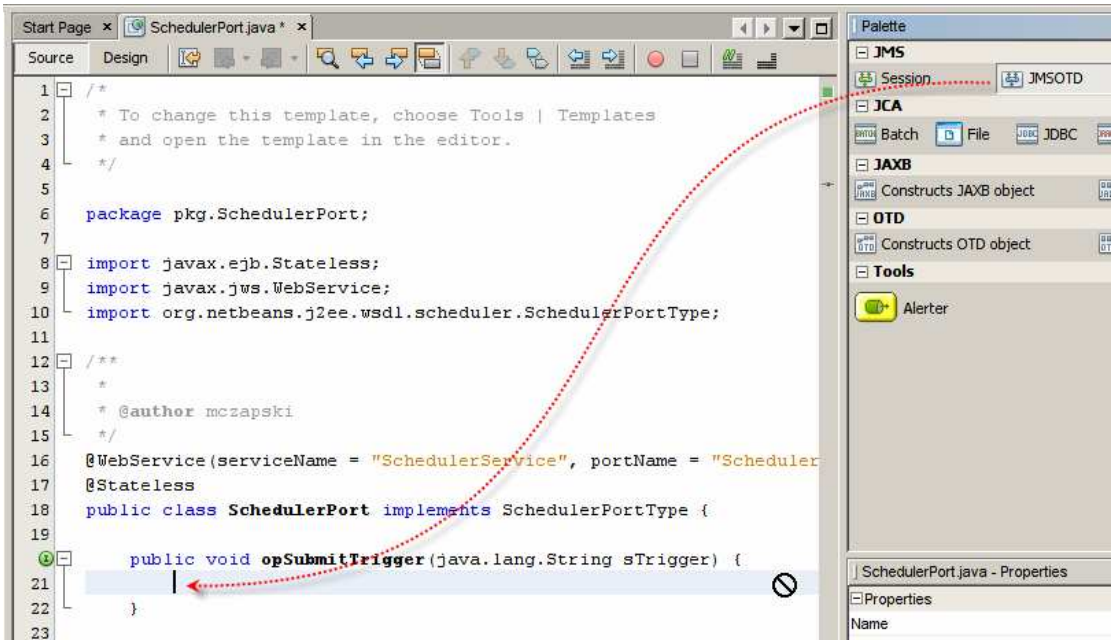

Figure 6-7 Start the JMSOTD configuration Wizard

Change or enter values as shown in Figure 6-8. At minimum provide the name of the Queue to which to queue the trigger message. Notice that we are not creating the Queue resource in the Application Server ahead of time nor are we bothered with JNDI references and such. The default JMS Message Server, when STCMS in installed as part of Java CAPS 6 installations, is STCMS. The queue (destination) we specify here will be automatically created when this application is deployed. Had we had a JNDI reference to a JMS Admin Object Resource we could have specified the JNDI name instead. It would have been something like lookup://jms/qSchedulerTrigger or similar, where “jms/qSchdulerTrigger” would be the JNDI name.

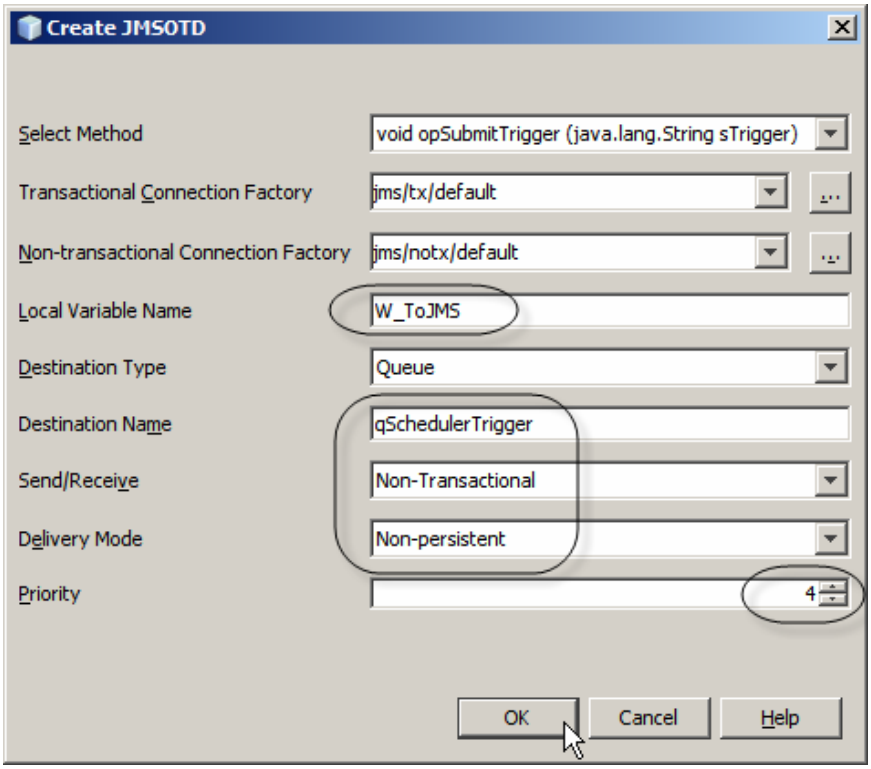

Figure 6-8 Configuring the JMSOTD

The wizard adds some boilerplate code, which, after reformatting, looks something like that shown in Figure 6-9.

```
18   public class SchedulerPort implements SchedulerPortType {
19
20       public void opSubmitTrigger(java.lang.String sTrigger) {
21           com.stc.connectors.jms.JMS W_ToJMS =
22                   com.stc.connectors.jms.JMS.createInstance
23                       (_jms_connfact
24                       , _notx_jms_connfact
25                       , com.stc.connectors.jms.JMS.DestinationType.Queue
26                       , "qSchedulerTrigger"
27                       , false
28                       , javax.jms.DeliveryMode.NON_PERSISTENT
29                       , 4);
30
31       }
32       _jms_connfact resource declaration. Click on the + sign on the left t
35       _notx_jms_connfact resource declaration. Click on the + sign on the
38
39   }
```

Figure 6-9 Boilerplate code added by the wizard, reformatted

To sent the trigger message to the Queue let’s add a few lines of code as shown in Figure 6-10.

```
public void opSubmitTrigger(java.lang.String sTrigger) {
    com.stc.connectors.jms.JMS W_ToJMS =
            com.stc.connectors.jms.JMS.createInstance
                (_jms_connfact
                , _notx_jms_connfact
                , com.stc.connectors.jms.JMS.DestinationType.Queue
                , "qSchedulerTrigger"
                , false
                , javax.jms.DeliveryMode.NON_PERSISTENT
                , 4);

    W_ToJMS.setTimeToLive(10 * 60 * 1000);
    try {
        W_ToJMS.sendText(sTrigger);
    } catch (JMSException ex) {
        Logger.getLogger
            (SchedulerPort.class.getName()).log(Level.SEVERE
            , "Exception sending trigger to Queue"
            , ex);
    }
```

Figure 6-10 Set time to live and send the text message to the Queue

The JCA-based Listener is done. Let's build and deploy it.

We have a number of ways to test the Listener. The simplest is to have NetBeans generate a Web Service client and us it to test the service. Figure 6-11 illustrates how the web service client can be produced.

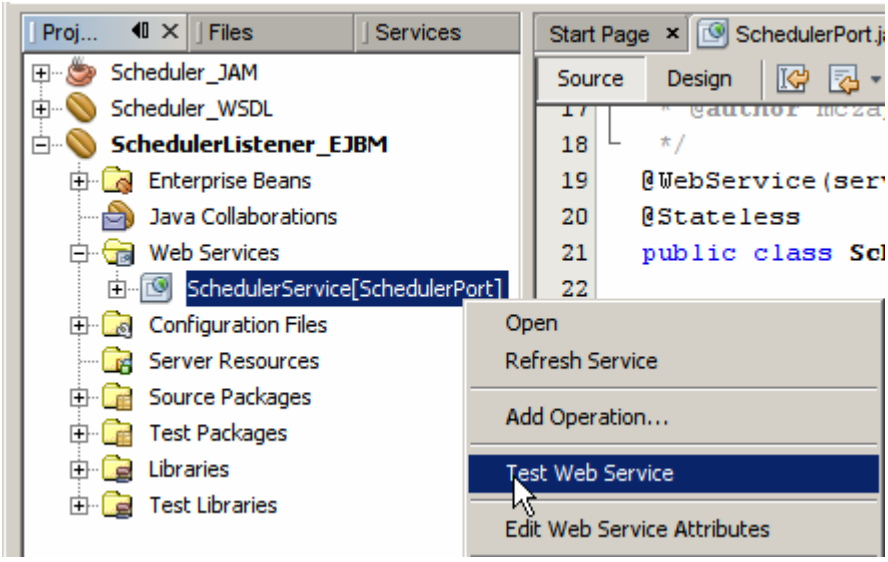

Figure 6-11 Getting a Web Service Client to test the service

Once the client code is generated your default web browser will appear with a web page similar to that shown in Figure 6-12. Enter some text and press the opSubmitTrigger button.

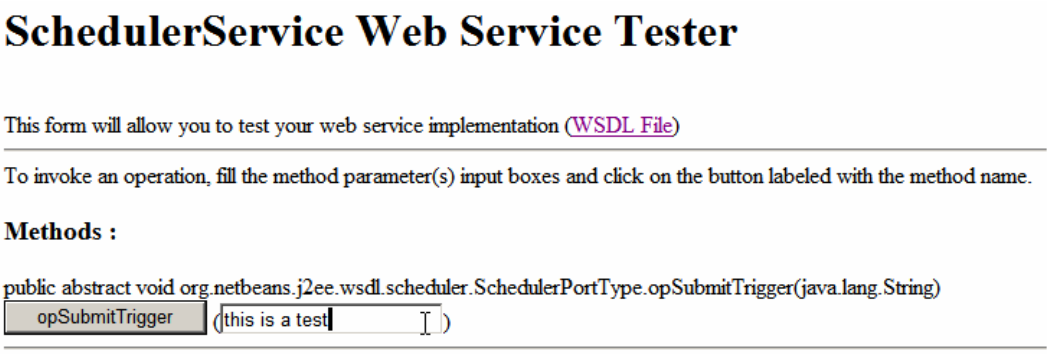

Figure 6-12 Web Service client servlet

Because the service implemented a One-Way Operation there will be no response. The response page will look similar to what is shown in Figure 6-13.

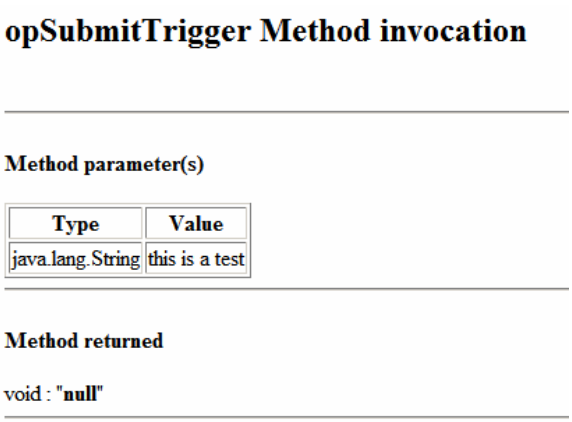

Figure 6-13 Response page

Let's open the Java CAPS 6 Enterprise Manager console and find out whether we have the message in the queue. Figure 6-14 illustrates the result.

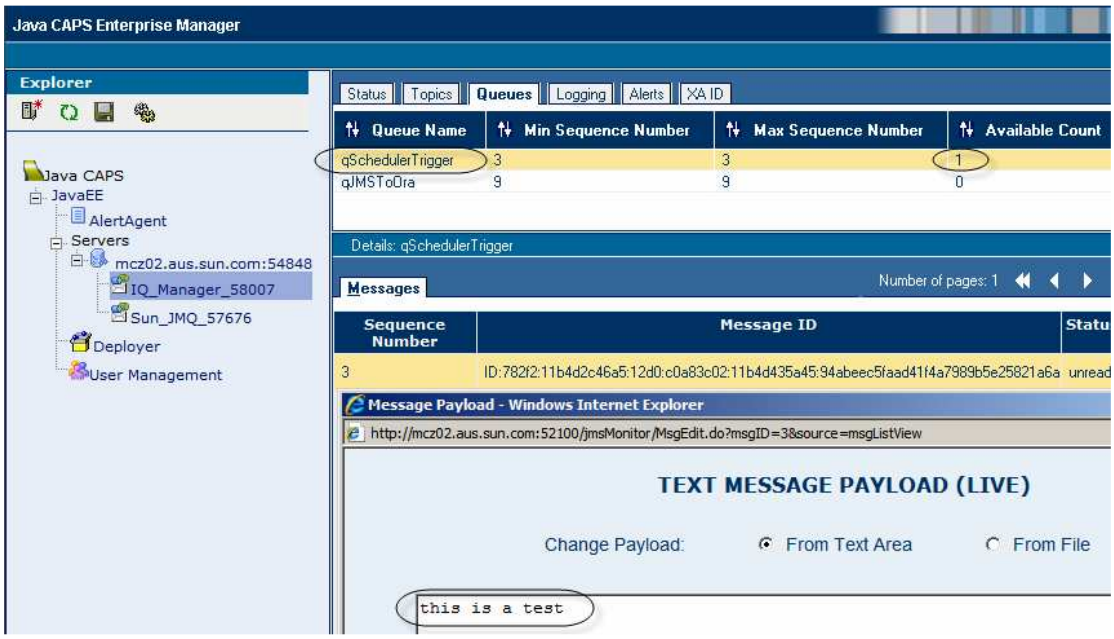

Figure 6-14 Message delivered to the Queue.

Let’s now exercise the Listener using the stand-alone client we developed in the previous section.

Let’s open a command prompt/DOS Box and change the working directory to where the application JAR file is located. Once there, let’s execute the command shown in previous section, and reproduced here:

```
c:\jdk1.6.0_02\bin\java -Djava.endorsed.dirs=.\lib -jar
Scheduler_JAM.jar localhost 58080 Hi there "Hello World"
```

Figure 6-15 shows the command and its output.

Figure 6-15 Command and its output.

Again, let’s use the Enterprise Manager to see if the trigger message made it into the Queue. Figure 6-17 shows the result.

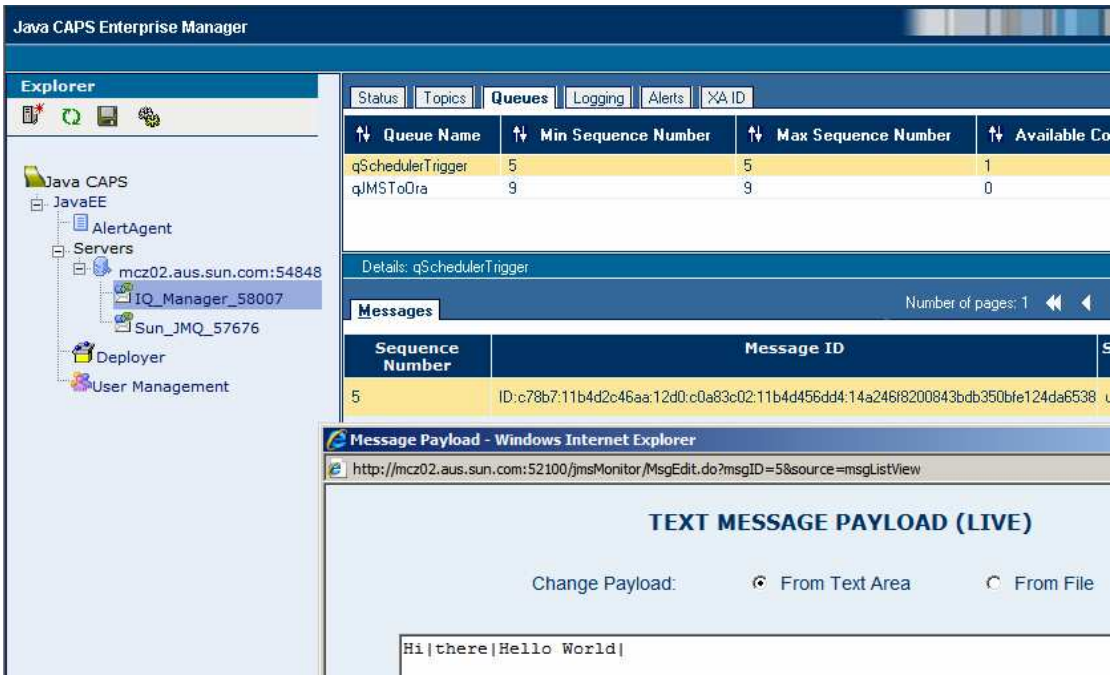

Figure 6-16 Trigger submitted by the stand-alone client application

7 Creating a JBI Trigger Receiver

In this section we will implement a web service that will accept the trigger string and deposit it in the hard-coded JMS Queue. As a JBI-based solution the implementation will use the SOAP Binding Component and the JMS Binding Component. Since there is no requirement for transformation logic we will develop a Composite Applicatin that connects the two Binding Components to each other through the NMR without any logic component.

Let’s create a SOA -> Composite Application Module, SchedulerListener_CAM, as shown in Figures 7-1 and 7-2.

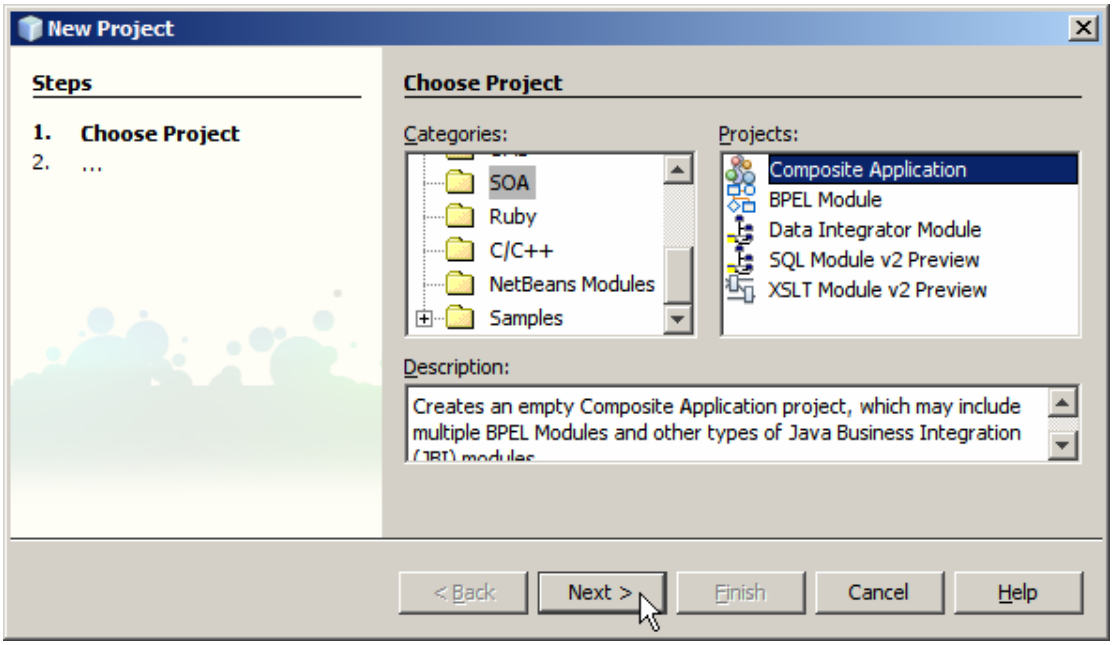

Figure 7-1 Start Composite Application Wizard

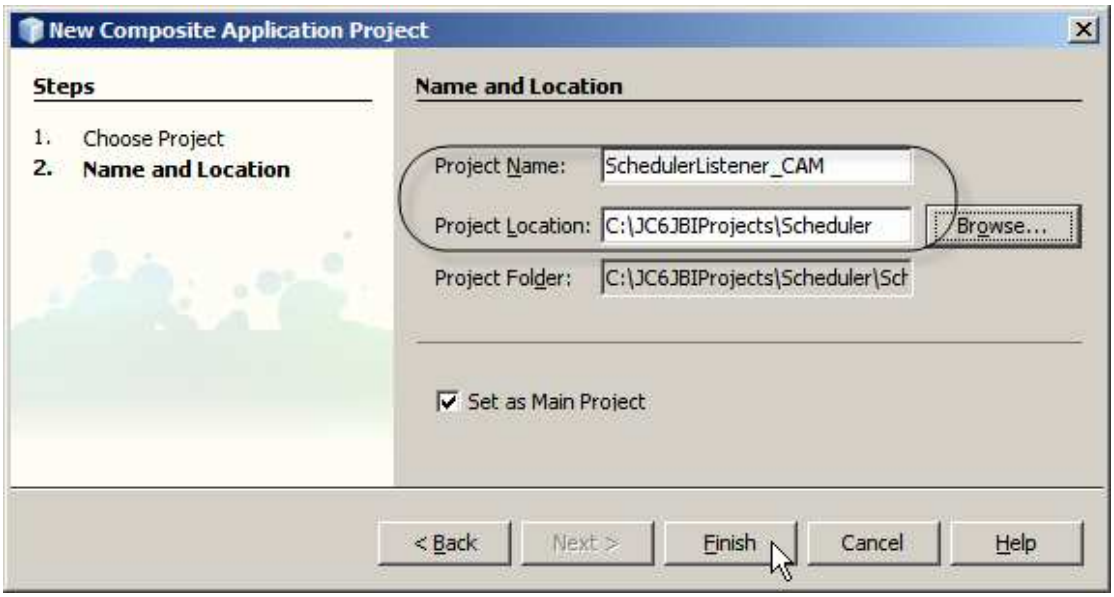

Figure 7-2 Name the project and choose the location

Let’s add the WSDL document from Scheduler_WSDL project so we can load it into the Service Assembly. Figures 7-3 and 7-4 illustrate major steps in the process.

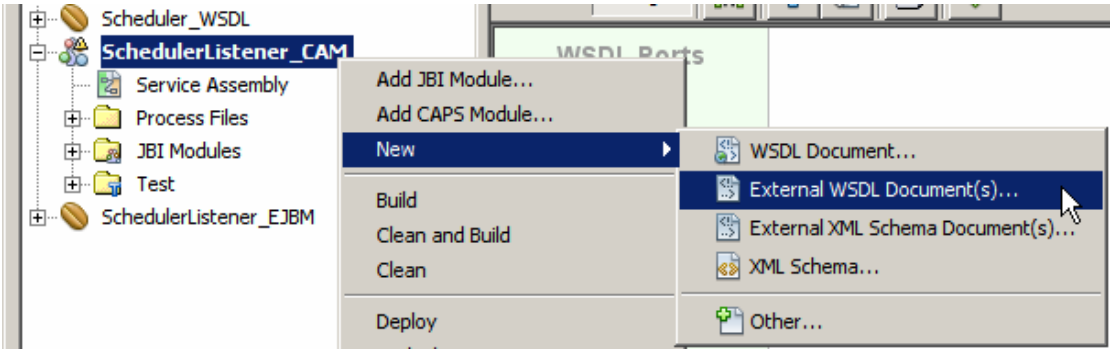

Figure 7-3 Start the Add External WSDL Wizard

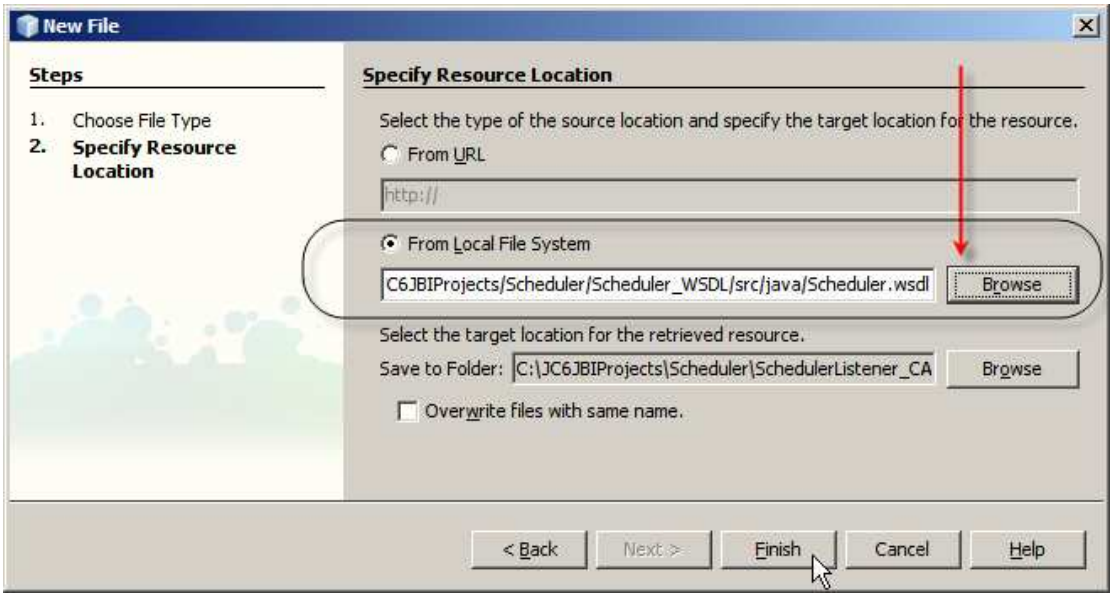

Figure 7-4 Locate the WSDL and Finish

Right click on the area of the WSDL Ports swim line and “Load WSDL Port”, as shown in Figures 7-5 and 7-6.

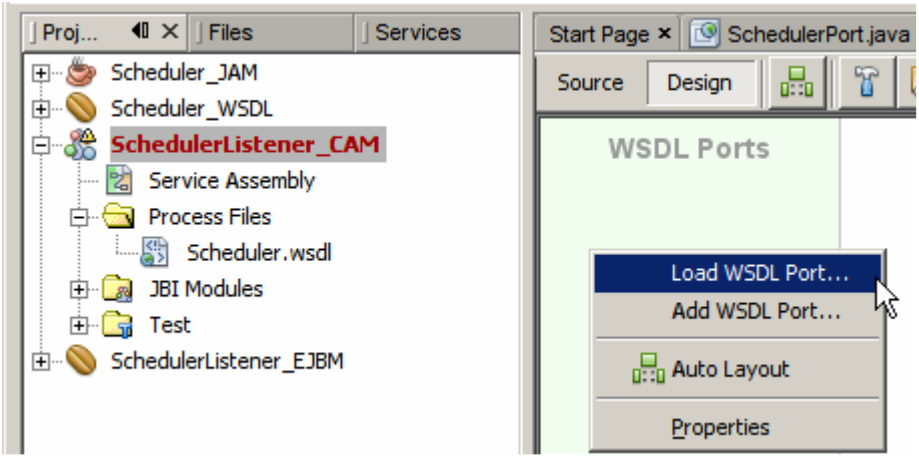

Figure 7-5 Start the Load WSDL Port Wizard

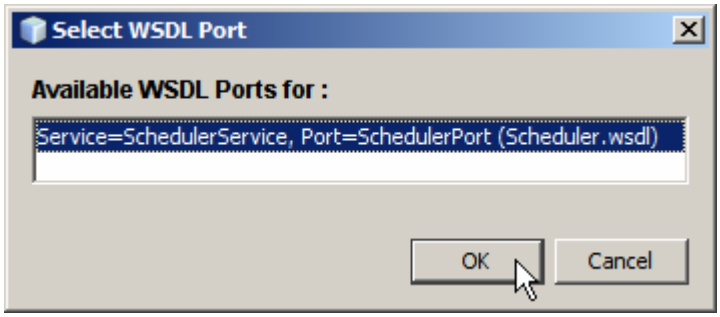

Figure 7-6 Choose the WSDL Port

Once the WSDL is loaded a new SOAP BC will be shown on the Service Assembly Editor canvas. To complete the Composite Application we need to drag the JMS BC form the palette, connect and configure. Figures 7-7 and 7-11 illustrate the steps.

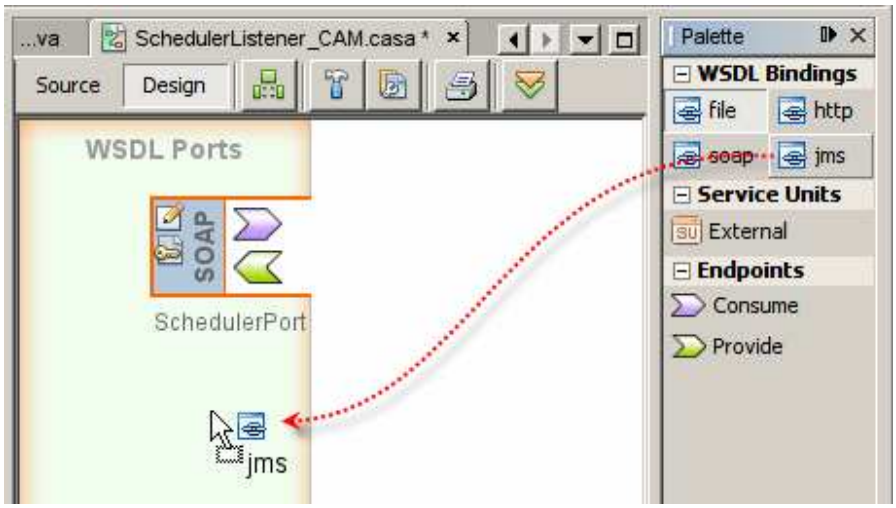

Figure 7-7 Drag JMS BC to the WSDL Ports swim line

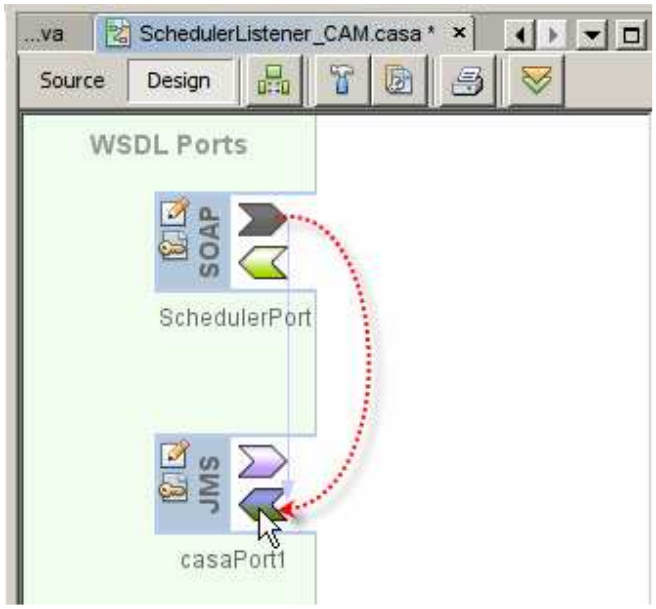

Figure 7-8 Connect by dragging form the Consume to the Provide icons

Once the JMS BC is added to the Service Assembly a corresponding WSDL, with the name derived from the name of the Composite Application, is created in the Process

Files folder. Open that WSDL, SchedulerListener_CAM.wsdl, and configure the jms:address node's properties to use stcms://<host>:<port> and admin/adminadmin credentials, as shown in Figure 7-9. If STCMS is installed it will be the default JMS Server. Since we used the default JMS server in our EJB-based listener, discussed in the previous section, we will use the STCMS in this section as well. This is to demonstrate that trigger messages go to the same Queue regardless of which listener, JCA-based or JBI-based, received and queued them.

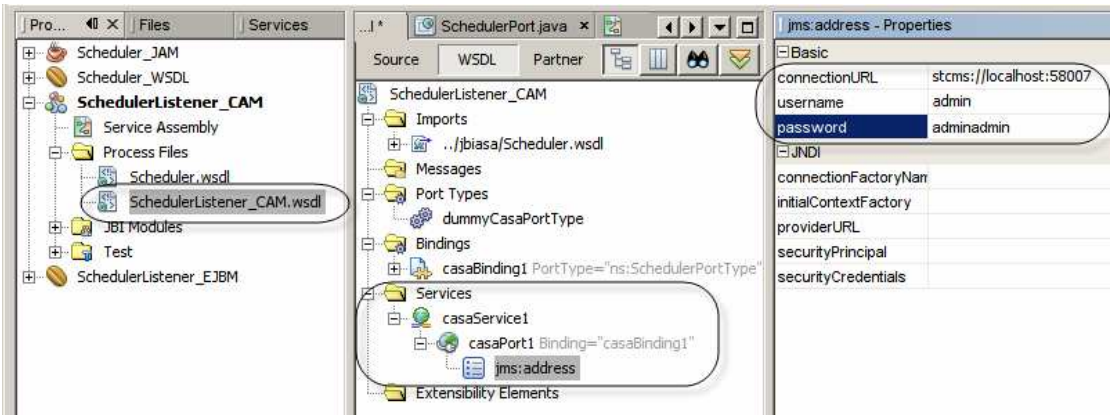

Figure 7-9 Configure jms:address properties in SchedulerListener_CAM.WSDL

Configure jms:operation properties, destination, transaction and deliveryMode, to be the same as what we configured for the JCA-based listener. Figure 7-10 shows the values.

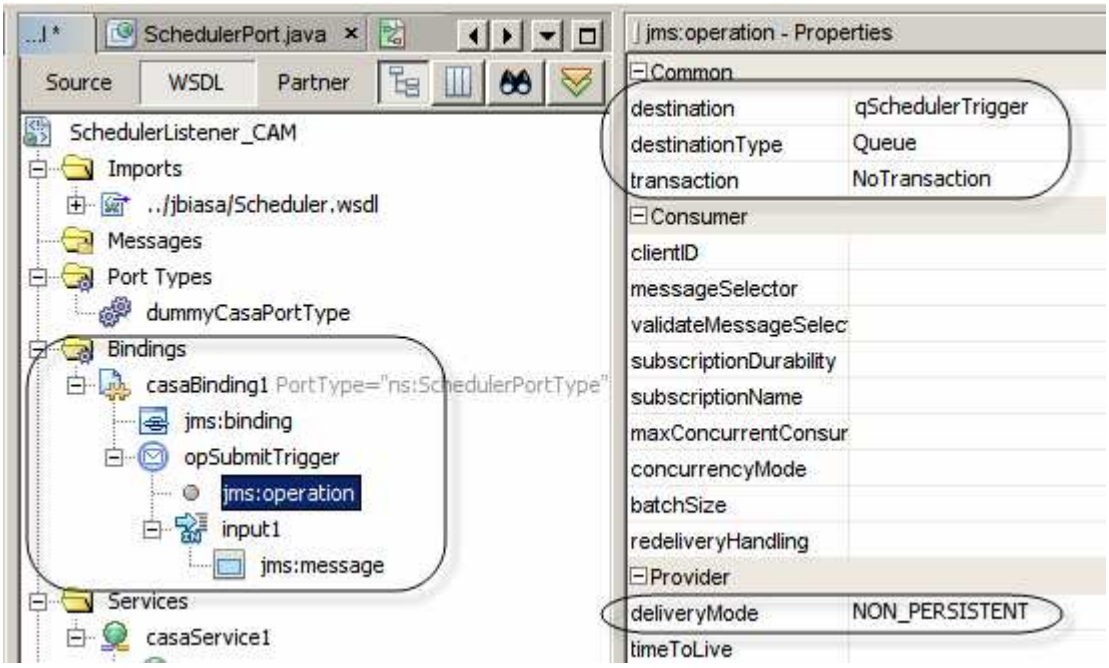

Figure 7-10 Configure jms:operation properties

Finally, we need to configure jms:message properties of interest. Figure 7-11 illustrates this. Recall that sTrigger is the name of the message part in the SOAP WSDL, which we created way back in section 3.

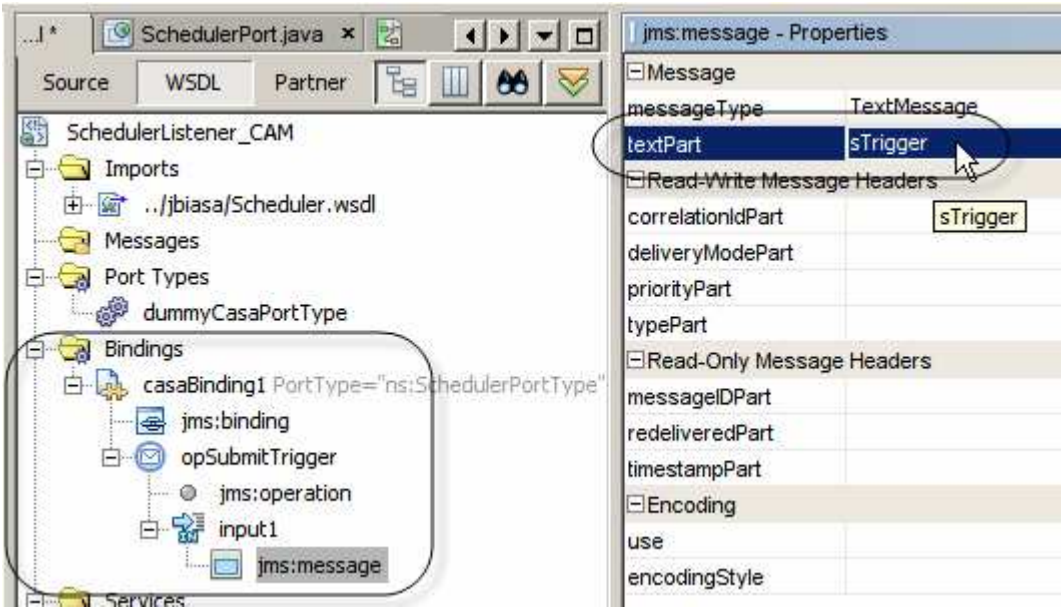

Figure 7-11 Configuring jms:message properties of interest

Since the WSDL is configured to use port 58080, in my case, and the JCA-based Listener is deployed and uses that port, we must change the address property of the SOAP BC. We will use a different, unused port number, say 58081. Figure 7-12 illustrates this.

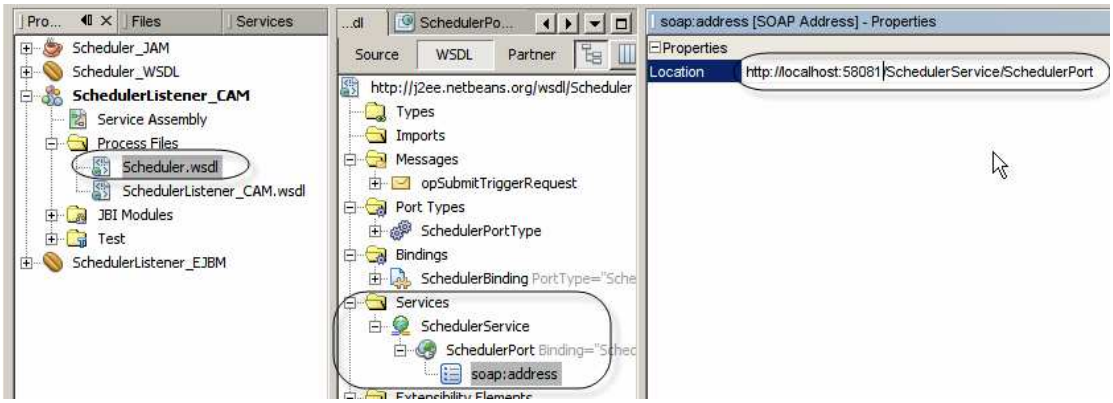

Figure 7-12 Changing end point port

Let's build and deploy the service.

Let's exercise the solution using the stand-alone client we developed in section 5. Figure 7-13 illustrates the command and its output.

```
C:\JC6JBIProjects\Scheduler\Scheduler_JAM\dist>c:\jdk1.6.0_02\bin\java -Djava.en
dorsed.dirs=.\lib -jar Scheduler_JAM.jar localhost 58081 Hi JBI "Hello World"
Host:   localhost
Port:   58081
Extra:  Hi¦JBI¦Hello World¦

C:\JC6JBIProjects\Scheduler\Scheduler_JAM\dist>
```

Figure 7-13 Exercising the JBI Listener

The result is a trigger message in the qSchedulerTrigger. Figure 7-14 shows the message.

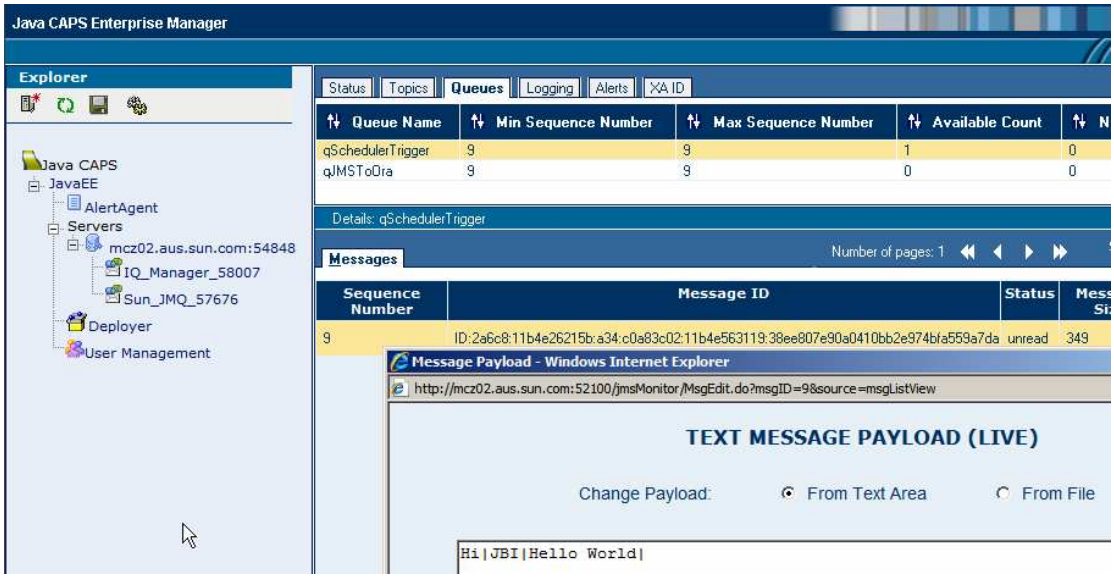

Figure 7-14 Trigger message in the queue

8 Windows Scheduled Tasks Scheduler (example)

Regardless of whether we use the stand-alone client with the JCA-based or the JBI-based listener we need to schedule its execution. On Windows we have the task Scheduler. This section gives an example of how a trigger can be scheduled using this environment. If you are using non-Windows environment use whatever scheduling facilities it has on offer. For example, for help on using Unix cron see you Unix system administrator or one of any number of online resoureces.

Let's imagine that we need to trigger a Java CAPS 6 or an OpenESB solution every day, every 5 minutes. We deliver a trigger which incorporates the date and time stamp, just to prove the command is executed and the trigger text can vary.

The command to execute will be:

```
cmd /c c:\jdk1.6.0_02\bin\java.exe -Djava.endorsed.dirs=.\lib -jar
Scheduler_JAM.jar localhost 58081 Hi JBI "Hello World" "%date%
%time%"
```

Note the %date% %time%. This construct will be replaced by the CMD shell with the date and time at the time the command is executed.

Notice, too, the relative path to the lib directory and the client JAR itself.

Let's navigate to Desktop\My Computer\Control Panel\Scheduled Tasks folder and double-click Add Scheduled Task application. Figure 8-1 illustrates this.

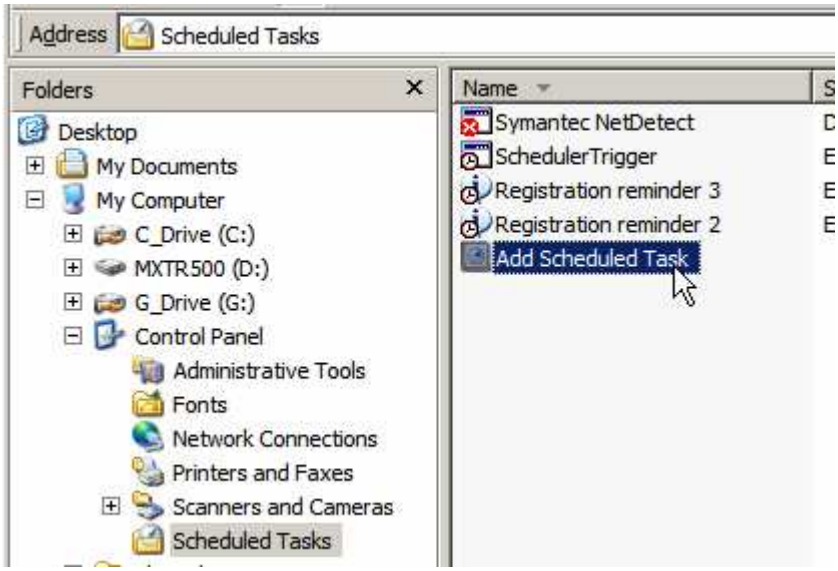

Figure 8-1 Starting Add Scheduled Task Wizard on Windows

In the following figures only key steps will be shown.

Locate the Java binary, see Figures 8-2 and 8-3.

Figure 8-2 Browse to locate the Java binary

Figure 8-3 Name the task SchedulerTrigger 02, choose "daily" and click Next

Accept the default or change them as you see fit. By default the task will be scheduled to start "now", whenever "now" is. Figure 8-4 shows my "now".

Figure 8-4 Accept defaults and click Next

Provide credentials for the user under whose control the task will be executed. Figure 8-5 illustrates this.

Figure 8-5 Provide user credentials

Check “Open Advanced Properties” and click Finish as shown in Figure 8-6.

Figure 8-6 Choose to configure advanced properties

Replace the java binary’s path with the entire command line, as shown at the beginning of this section, specify the directory path of where the Scheduler_JAM.jar, the stand-alone client, is located and click Apply. Figure 8-7 illustrates this.

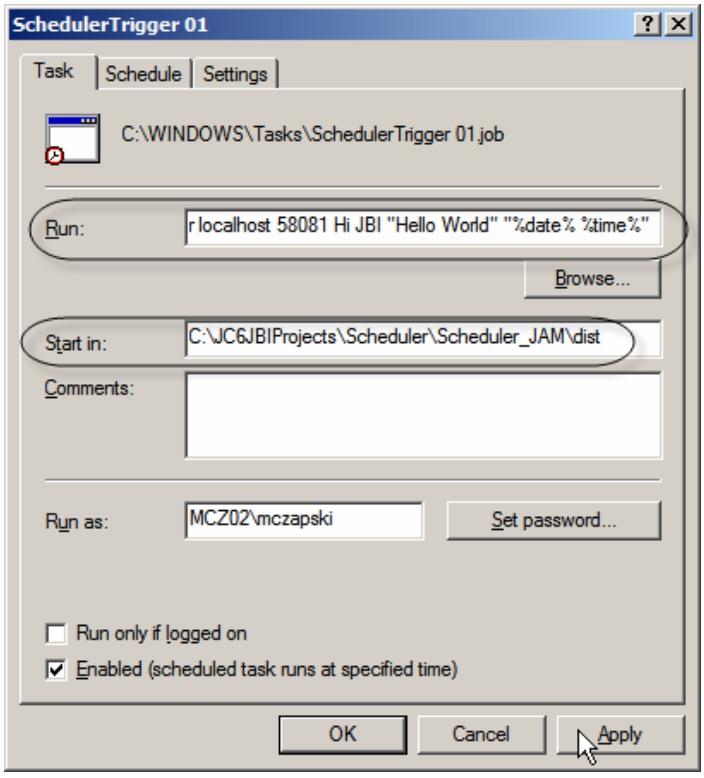

Figure 8-7 Configure the command and the working directory

Click “Schedule” tab, click “Advanced” button and configure task schedule details as shown in Figure 8-8.

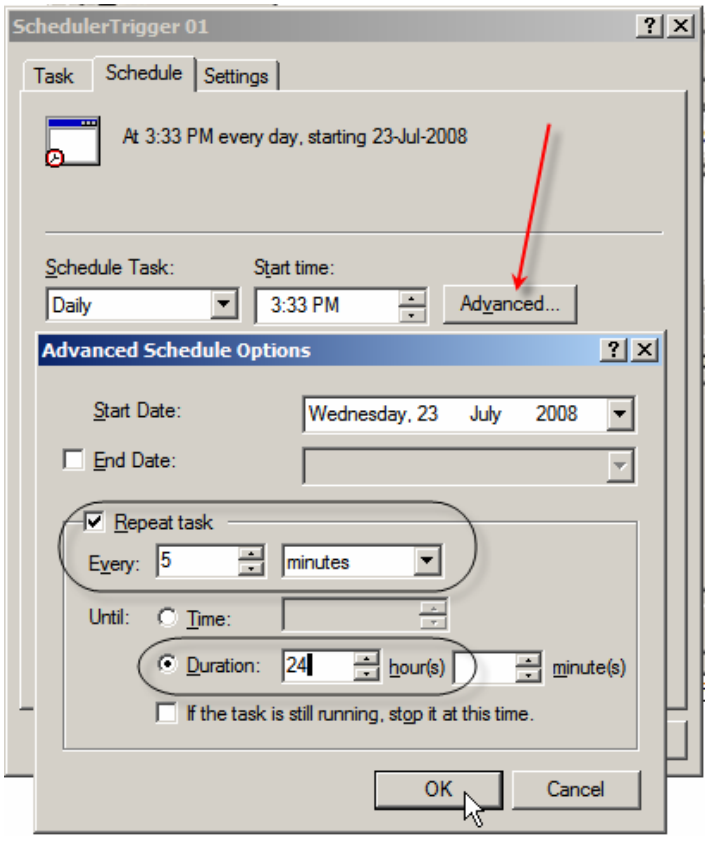

Figure 8-8 Configure task to run every 5 minutes for ever

Complete the dialog.

You should see a task line that reads something like what is shown in Figure 8-9.

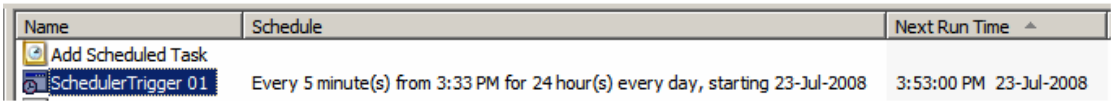

Figure 8-9 Configured Task

Within at most 5 minutes the task should be scheduled for execution and should begin.

If the task executed successfully it will be re-scheduled and the task line will read something like what is shown in Figure 8-10.

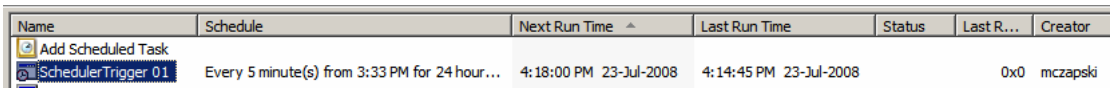

Figure 8-10 Task executed successfully and re-scheduled

The Enterprise Manager will show triggers queued up. Figure 8-11 shows an example.

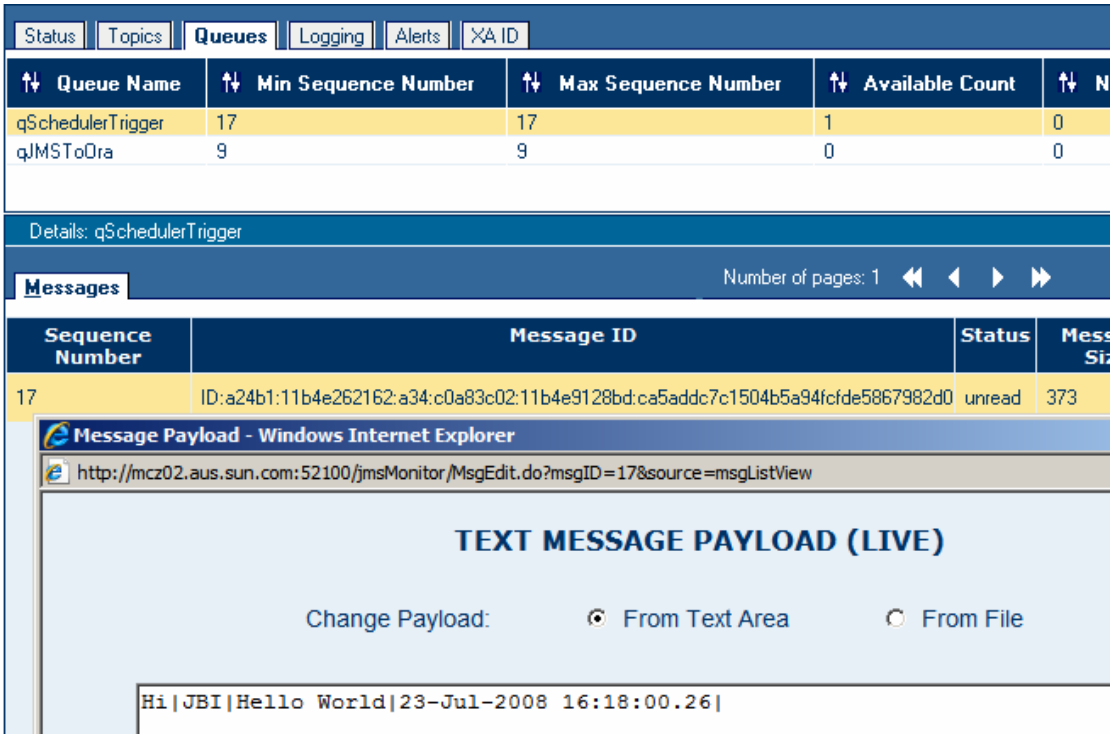

Figure 8-11 Example trigger with date/time stamp embedded

If the task failed see the Task Scheduler log to try to determine what went wrong. Figure 8-12 points out how to get at the task scheduler log.

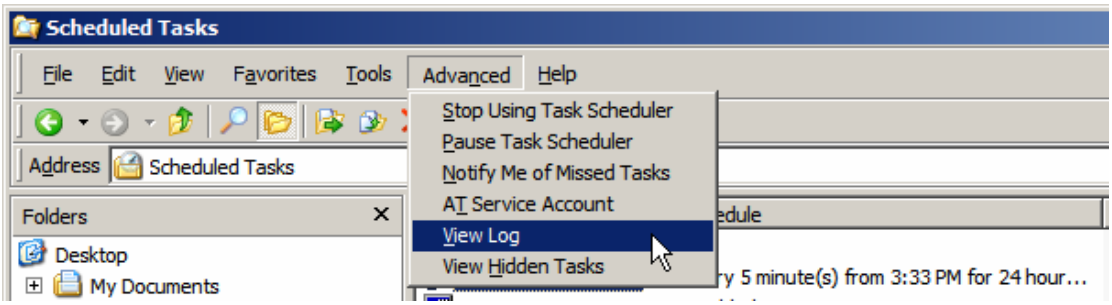

Figure 8-12 See Task Scheduler log

Once you are happy that the experiment works, you can delete the task.

9 Potential Improvements

Some potential improvements or variations on the subject present themselves.

The client application could use a multi-part or structured message, instead of a simple string. Message parts could be used to convey the name of the sender application, timestamp of the trigger action, name of the queue to which to queue the trigger, trigger time-to-live value and other data items that could be varied between trigger instances and that could modify processing behavior at the server side.

Rather then having a separate server to receive trigger messages and queue them to a JMS Queue the actual business solution could implement the web service receiver and be directly triggered by the trigger message. This would eliminate JMS as the intermediary.

The reader can, I am sure, think of other variations that will better suit their requirements.

10 Summary

This document walked the reader, step-by-step, through the process of creating, building, deploying and exercising a Java CAPS 6 and OpenESB solution that can be used to schedule tasks and that can be used to schedule other solutions deployed to the JavaEE or the JBI containers.