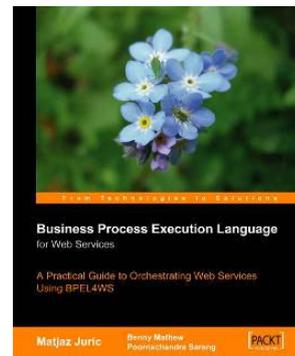




Business Process Execution Language for Web Services

Matjaz B Juric
Benny Mathew
Poornachandra Sarang



Chapter 4 “Oracle BPEL Process Manager”

For more information: www.PacktPub.com/book/BPEL

In this Package, you will find:

- A Biography of the authors of the book
- A preview chapter from the book, chapter 4 “Oracle BPEL Process Manager”
- A synopsis of the book’s content
- Information on where to buy this book

About the Authors

Matjaz B. Juric holds a Ph.D. in computer and information science. He works for the University of Maribor. He has co-authored Professional J2EE EAI, Professional EJB, J2EE Design Patterns Applied, and VB.NET Serialization Handbook, published by Wrox Press. He has published chapters in the book *More Java Gems* (Cambridge University Press) and in *Technology Supporting Business Solutions* (Nova Science Publishers). He has also published in journals and magazines such as Java Developer's Journal, Java Report, Java World, Web Services Journal, eai Journal, ACM journals, and presented at conferences such as OOPSLA, SIGS Java Development, XML Europe, SCI, and others. He is also a reviewer, program committee member, and conference co-organizer. Matjaz has been involved in several large-scale object technology projects. In association with the IBM Java Technology Centre, he worked on performance analysis and optimization in RMI-IIOP development, an integral part of the Java 2 Platform. He has recently been classified in the Techindex Evangelist.

My efforts in this book are dedicated to my family. Special thanks to Jerneja, my friends at University of Maribor, and to Louay at Packt Publishing.

Benny Mathew is a Sr. Software Engineer at Hewlett-Packard (Global Delivery India Center). He holds a Masters degree in Computer Applications. His fascination for computers dates back to high school days and he has been programming with a passion for more than a decade and a half. He has also co-authored *Visual Basic .NET Reflection Handbook* published by Wrox press. During his free time, Benny likes to write technical articles and help people on the newsgroups relating to .NET technologies and has been awarded Microsoft Most Valuable Professional (MVP) for two consecutive years now. Before joining Hewlett Packard, he was with companies like Thomson Financials and Delphi Software in Bangalore India. You can reach him at benny@myps.org.

I'd like to thank Girish Nadkarni for recommending my name to the editor of this book, and to Louay Fatoohi and all the reviewers; they really helped to steer my writing in the right direction. I'd also like to thank my family for their support in writing this book.

Poornachandra Sarang, Ph.D., is CEO of ABCOM Information Systems. He has been a Visiting Professor of Computer Engineering at University of Notre Dame, USA and is currently a visiting professor for Post-Graduate Computer Science courses at University of Mumbai. Dr. Sarang provides consulting services to worldwide clients in architecting and designing IT solutions based on Java, CORBA, and Microsoft platforms. A well known and a highly sought after trainer, Dr. Sarang has conducted several training programs on latest technologies for several top-notch IT companies. He conducts lectures/seminars on emerging technologies across the world and has made several presentations in international conferences. He has authored/co-authored several books on Java, C++, J2EE, e-Commerce, and .NET.

Oracle BPEL Process Manager

In this chapter we will take a detailed look at the Oracle BPEL Process Manager Version 2.0, a BPEL server that enables us to deploy and run business processes defined in BPEL. Oracle BPEL Process Manager is developed in Java and runs on a J2EE application server. In addition to deploying and running BPEL processes, it offers advanced functionality that makes it one of the most powerful BPEL servers at the time of writing this book. Oracle also offers a BPEL Designer that enables BPEL process development using an intuitive graphical editor instead of writing BPEL code by hand and also allows us to automatically deploy BPEL processes. The BPEL Designer eases the development (and maintenance) of BPEL processes considerably.

While discussing the capabilities of the Oracle BPEL Process Manager and the BPEL Designer, we will discuss the following:

- Architecture of the BPEL Process Manager
- Major features
- Process deployment
- Managing and debugging processes with BPEL Console
- Graphical development with BPEL Designer
- Oracle-specific functions such as XSLT, XQuery, and XSQL engines
- Integration of BPEL processes with e-mail and JMS
- Integration with Java and J2EE
- XML business document façades
- Web Services Invocation Framework bindings
- Oracle BPEL Server Java APIs
- User tasks and their integration into processes

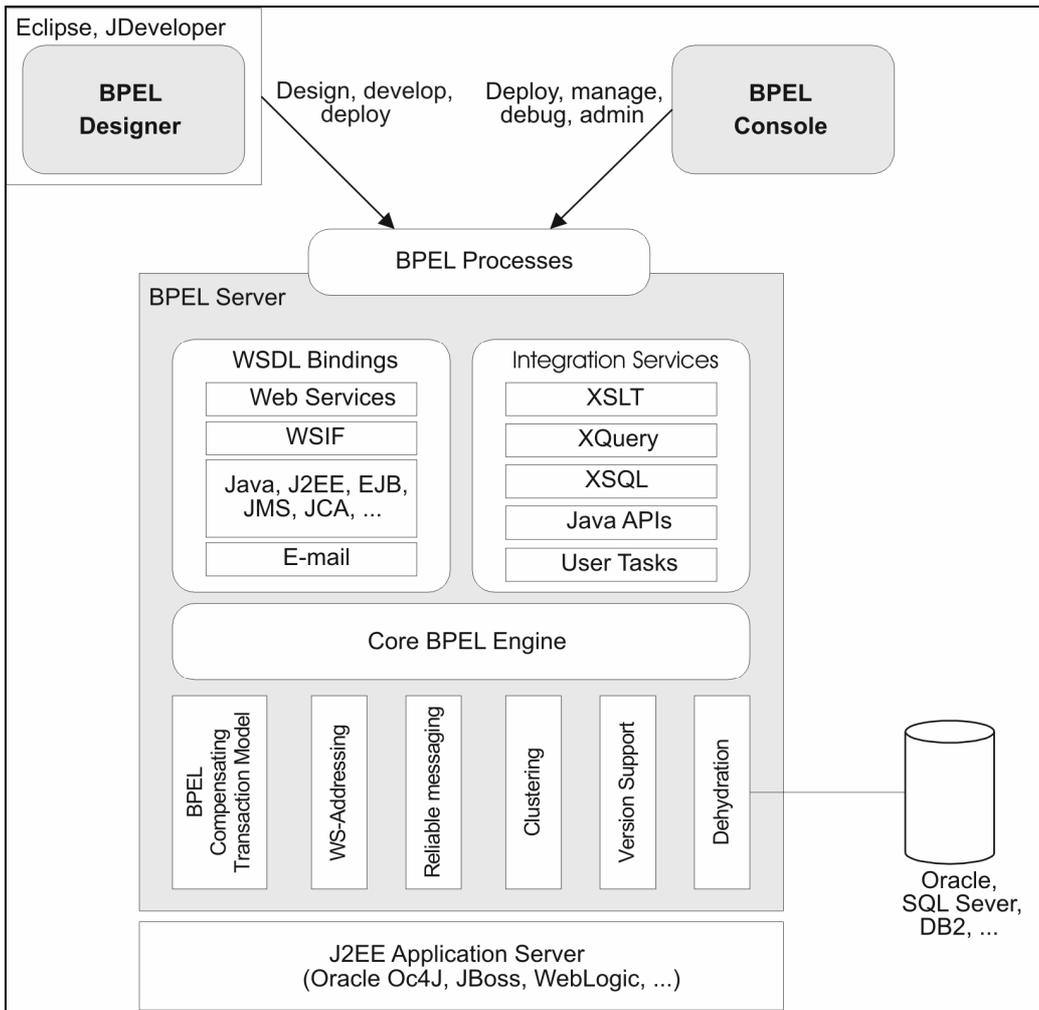
Overview and Architecture

The Oracle BPEL Process Manager (formerly known as Collaxa BPEL Server) is a run-time environment for BPEL processes. BPEL Process Manager 2.0 fully supports BPEL version 1.1 and provides additional tools for deployment, monitoring, and management of BPEL processes. At the time of writing this book, Oracle BPEL Process Manager is one of the most complete BPEL servers available.

BPEL Process Manager is developed in Java and runs on a J2EE-compliant application server, for example, the Oracle Application Server OC4J (Oracle Containers for Java). In addition to the OC4J version, Oracle also provides versions for the open source JBoss and for the BEA

WebLogic Server. With manual installation, Oracle BPEL Process Manager can also be used with IBM and Sun application servers.

Let us now look at the architecture of the BPEL Process Manager, shown in the following figure:



The Oracle BPEL Process Manager has four major parts:

- BPEL Designer
- BPEL Server
- BPEL Console
- Database

BPEL Designer

BPEL Designer enables us to develop BPEL processes visually in a graphical environment without having to write BPEL code by hand. Instead, we can drag and drop activities to the process. We can add partner links and locate services through the UDDI browser. We can also use function and copy wizards. BPEL Designer can deploy the developed processes directly to the BPEL Server. This eases the development and maintenance of BPEL processes considerably. BPEL Designer is a plug-in for the Eclipse 3.0 platform, but we expect that it will become a part of the Oracle JDeveloper soon. Because it uses standard BPEL, processes developed by the BPEL Designer can be used with other BPEL servers (and vice-versa) as long as we do not use functionality specific to the Oracle product. We will discuss BPEL Designer later in this chapter.

BPEL Server

BPEL Server runs in a J2EE-compliant application server. It has the following main parts:

- Core BPEL engine
- WSDL bindings
- Integration services

Core BPEL Engine

The core BPEL engine is the run-time environment where the BPEL processes are deployed and executed. In addition to full BPEL v1.1 support, the engine provides support for key web services orchestration stack technologies, particularly WS-Addressing and the BPEL compensating transaction model.

The BPEL engine also provides support for version control. This enables us to develop several versions of a business process and deploy them side by side. This feature is important in real-world scenarios because business processes evolve over time. Having an effective versioning support simplifies the management.

Another very important feature is **dehydration**. In previous chapters we explained that business processes can be long-running because the involved partners might not be able to react instantly to the requests. This happens particularly in asynchronous scenarios where a business process invokes a partner web service (using the `<invoke>` activity) and then waits for the response (using the `<receive>` or `<pick>` activities). While waiting for the response the Oracle engine can store the process (and its state) in the database, thus freeing up server resources. This is called dehydration. When the engine receives the response it first restores the process with its state from the database (**hydration**) and then continues with the execution of the process. In real-world

scenarios where many business processes might be running side by side, the dehydration capability is important as it reduces the demands on hardware performance.

Oracle BPEL engine also provides support for clustering. **Clustering** increases server reliability because fail-over can be configured on the engine. Clustering also improves scalability with load balancing. These features are very important in real-world usage of the product.

WSDL Bindings

The WSDL binding framework is responsible for communication with the BPEL processes deployed on the server. This includes clients that would like to access a BPEL process and BPEL processes that would like to access other web services (partner links). Although the BPEL specification talks only about web services, the Oracle BPEL Server even enables connectivity using protocols other than SOAP. In real-world scenarios, a business process will often have to connect to an existing application or system. Using the WSDL binding framework, the reach of BPEL is extended to systems using protocols other than those supported by web services (primarily SOAP).

Of particular interest here is connectivity to J2EE artifacts, such as EJBs (Enterprise Java Beans), RMI (Remote Method Invocation), JMS (Java Message Service), JCA (Java Connector Architecture), and also to e-mail and HTTP GET and POST. The integration is achieved through the **WSIF** (Web Services Invocation Framework) from Apache (<http://ws.apache.org/wsif/>). All this enables relatively easy and effective integration of backend systems, particularly existing and legacy systems, which cannot be simply exposed as web services.

Integration Services

Business processes described in BPEL communicate with web services and exchange XML documents. The integration services enable us to perform transformations (on these XML documents) that go beyond the support of XPath.

Oracle BPEL Server provides support for XSLT transformations, XQuery, and XSQL.

XSLT (Extensible Stylesheet Language for Transformations) provides support for complex transformations of XML vocabularies and can also be used to transform XML to other markup formats such as HTML, WML, or VoiceXML for presentation purposes. For more information on XSLT please refer to <http://www.w3.org/TR/xslt>. XQuery and XSQL are XML query languages with functionality that goes beyond simple XPath queries. For more information on XQuery please refer to <http://www.w3.org/XML/Query>. For more information on XSQL please refer to Oracle documentation.

BPEL Server also provides Java integration. We have two choices:

- We can embed Java code in BPEL processes.
- We can use the Web Services Invocation Framework (WSIF).

The BPEL server exposes its functionality through a set of APIs. An important part of the integration services is the **user task service**. The built-in BPEL service provides an easy way to include user interaction in BPEL processes. Business processes often require that a user reviews or

confirms a decision before carrying out further steps. However, the BPEL specification does not provide an easy way for doing this. The Oracle BPEL Server therefore provides user tasks through which we can include user interaction in an easy way, as we will see later in the chapter.

BPEL Console and Database

Through BPEL Console we can deploy, manage, administer, and debug BPEL processes. The most important features of the BPEL Console include:

- Visual process flows
- Audit trails
- Debugging view of processes
- Process history

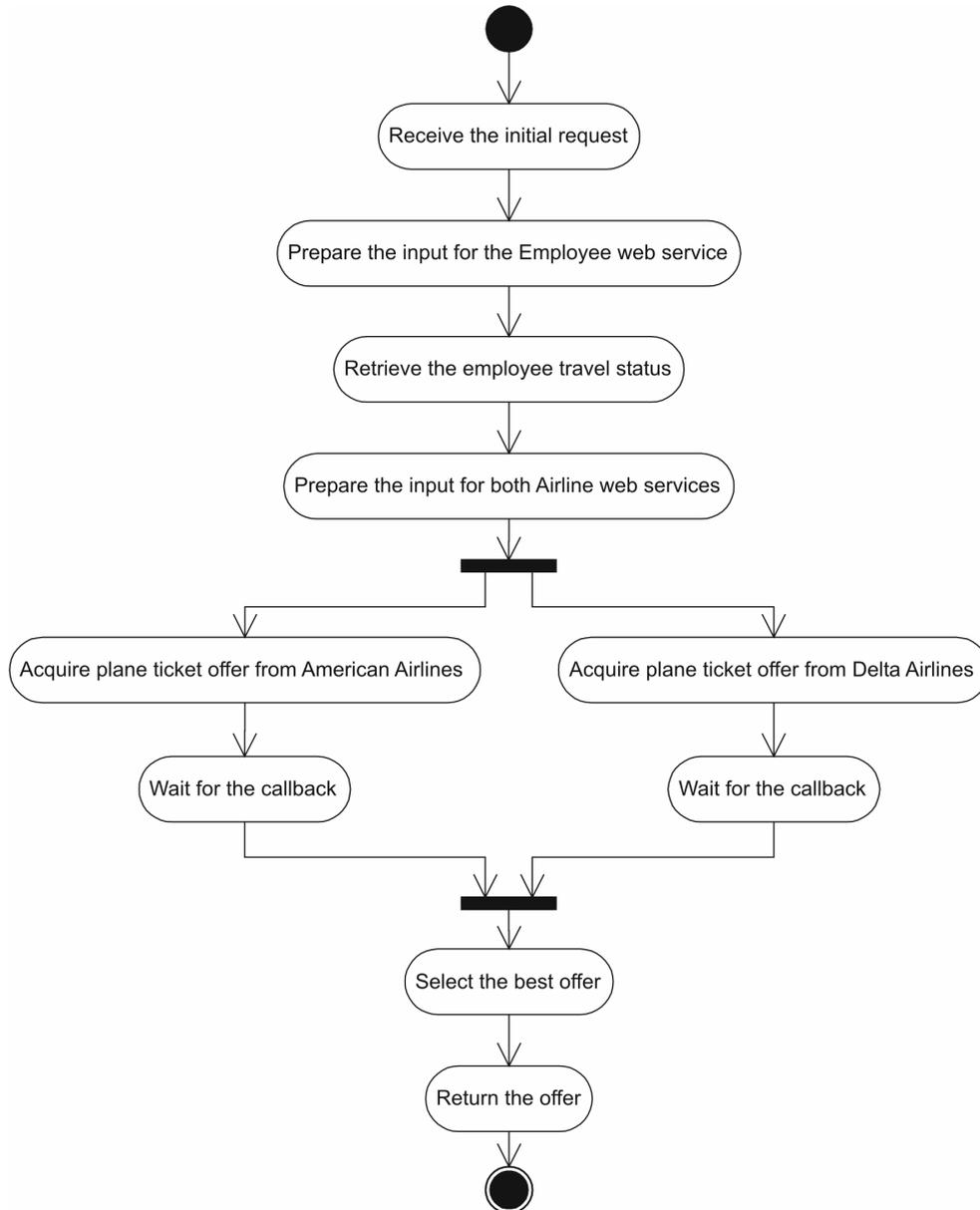
Oracle BPEL Console uses a web-based interface, which is basically a set of JSP (Java Server Pages) and servlets that call the BPEL Server API (in Java). This means we could easily develop our own console if we need specific handling of BPEL processes.

We have already mentioned that Oracle BPEL Server supports dehydration, which stores the process state in the database. BPEL Server supports Oracle DBMS, Microsoft SQL Server, and IBM DB2. It can actually be configured to use any JDBC database. The trial version, which can be downloaded from Oracle web site, comes bundled with Oracle Lite. Note that for real-world scenarios a production-quality database should be used.

Process Deployment Example

Let us now show how we deploy a BPEL process on the Oracle BPEL Server. We will assume that Oracle BPEL Process Manager has been successfully installed according to the installation instructions and that it uses the default port 9700. If another port has been selected during installation, the examples have to be modified accordingly.

We will use the Business Travel BPEL process example that we developed in Chapters 2 and 3. The travel example is a simplified business process that selects the best airline ticket offer. To refresh our memory, let us have a look at the process activity diagram:



In the previous chapters we developed the BPEL code for the example; this consists of a `Travel.bpel` file with the source code and the `Travel.wsdl` where the WSDL definitions are stored. We will not show the source of these files here as they have been already shown in previous chapters. They can also be downloaded from <http://www.packtpub.com>.

Process Descriptor

Each BPEL process we deploy to the Oracle BPEL Process Manager requires a process descriptor. This process descriptor is not covered by the BPEL standard and is specific to the BPEL server.

The deployment process descriptor is the only part of the implementation of a process on a given platform that *must* be re-written to run the process on a different BPEL engine.

The Oracle process descriptor is an XML file specifying the following details about the BPEL process:

- BPEL source file name
- BPEL process name (ID)
- WSDL locations of all partner link web services
- Optional configuration properties

The default file name for the process descriptor is `bpel.xml`, but we can use any other name. Let us now write the process descriptor for our process. First we have to specify the XML header and the `<BPELSuitcase>` root element. In the `<BPELProcess>` element we specify two attributes, `src`, which denotes the BPEL source file name (`Travel.bpel`), and `id`, which denotes BPEL process name as shown in the BPEL Console (we will use the `TravelProcessCh4` ID):

```
<?xml version="1.0" encoding="UTF-8"?>
<BPELSuitcase>
  <BPELProcess src="Travel.bpel" id="TravelProcessCh4">
  ...
```

Next we specify the partner link binding properties for the location of the WSDL for each partner link that we use in the process. In our travel example process we use the following partner links:

- `client`: Used for client interaction with the process
- `employeeTravelStatus`: The link to the Employee web service
- `AmericanAirlines`: The link to the American Airline web service
- `DeltaAirlines`: The link to the Delta Airline web service

The WSDL for the `client` partner link is stored locally in the `Travel.wsdl` file. For the location of the other three partner web services' WSDLs, we have to specify the corresponding URLs. Here we have provided simplified implementations of all three web services, which can also be downloaded and deployed on the Oracle BPEL Server. The rest of the process descriptor with the location of the WSDLs is shown below:

```
... <partnerLinkBindings>
  <partnerLinkBinding name="client">
    <property name="wsdlLocation">
      Travel.wsdl
    </property>
  </partnerLinkBinding>
```

```

    <partnerLinkBinding name="employeeTravelStatus">
      <property name="wsdlLocation">
        http://localhost:9700/orabpel/default/Employee/Employee?wsdl
      </property>
    </partnerLinkBinding>

    <partnerLinkBinding name="AmericanAirlines">
      <property name="wsdlLocation">
        http://localhost:9700/orabpel/default/AmericanAirline/AmericanAirline?wsdl
      </property>
    </partnerLinkBinding>

    <partnerLinkBinding name="DeltaAirlines">
      <property name="wsdlLocation">
        http://localhost:9700/orabpel/default/DeltaAirline/DeltaAirline?wsdl
      </property>
    </partnerLinkBinding>
  </partnerLinkBindings>
  ...

```

Optionally we can add configuration properties such as introduction text outputted by the BPEL Console when starting the process and default input data. The introduction text should be included within the `<property>` element with the attribute `name` set to `testIntroduction`:

```

  ...
  <configurations>
    <property name="testIntroduction">
      The Business Travel Process example.
    </property>
  ...

```

To add the default input data (also optional) we have to define the `<property>` element with the attribute `name` set to `defaultInput` and provide the input XML message as `CDATA`:

```

  ...
  <property name="defaultInput">
    <![CDATA[
      <TravelRequest xmlns="http://packtpub.com/bpel/travel/">
        <employee xmlns="http://packtpub.com/service/employee/">
          <FirstName>Matjaz B.</FirstName>
          <LastName>Juric</LastName>
          <Departement>University</Departement>
        </employee>
        <flightData xmlns="http://packtpub.com/service/airline/">
          <OriginFrom>Ptuj</OriginFrom>
          <DestinationTo>London</DestinationTo>
          <DesiredDepartureDate>2004-04-20</DesiredDepartureDate>
          <DesiredReturnDate>2004-04-24</DesiredReturnDate>
        </flightData>
      </TravelRequest>
    ]]>
  </property>
  ...
</configurations>
</BPELProcess>
</BPELSuitcase>

```

Setting the Environment

We are now ready to start the BPEL Process Manager. We can do this from the `Start` menu (if using Windows) or by executing the `startOrabpel` script, which can be found in the

`c:\orabpel\bin` directory (assuming Oracle BPEL Process Manager has been installed in `c:\orabpel`). It is recommended that we include this directory in the path for easy access.

Next we will need a command prompt where we have to set the environment variables. We can do this by executing the `obsetenv` script in the same `c:\orabpel\bin` directory. The script sets the following environment variables:

- **OB_HOME**: Specifies the path to the Oracle BPEL installation directory (`c:\orabpel` is the default path)
- **OB_PLATFORM**: Specifies the application server (`oc4j_10g` if using Oracle OC4J)
- **MY_CLASSPATH** and **MY_CLASSES_DIR**: Specify the class path for Oracle BPEL Server
- **JAVA_HOME**: Points to the Java SDK home directory
- **OB_JAVA_PROPERTIES**: Specifies the WSDL factory and the options proxy settings
- **J2EE_APPLICATIONS**: Specifies the application server directory where J2EE applications can be deployed

Setting the environment variables is essential for successful deployment of BPEL processes.

BPEL Compiler

After we have written the process descriptor and set the environment we are ready to deploy the BPEL process. For this, Oracle BPEL Process Manager provides the **BPEL Compiler**, which can be started with the `bpelc` command from the command line. `bpelc` compiles the BPEL process source files and creates the BPEL process archive JAR file. It can also automatically deploy the process to the Oracle BPEL Server. This is discussed in the next section.

The BPEL compiler has the following syntax:

```
> bpelc [options] process_descriptor_name.xml
```

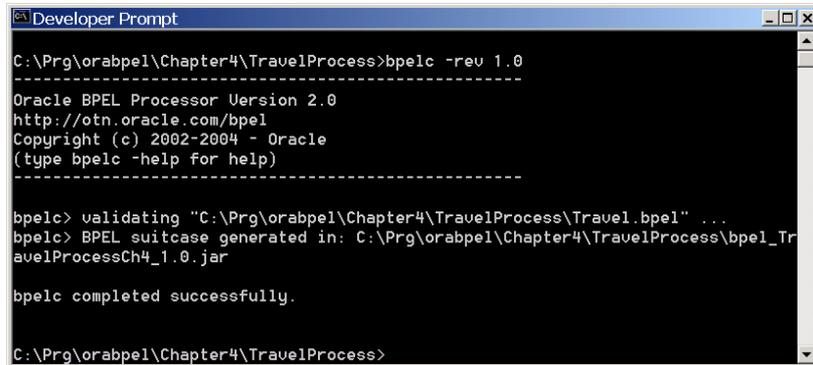
The default for `process_descriptor_name.xml` is `bpel.xml`. The most important options are:

- **-rev <revision_tag>**: Specifies the revision (version) number for the deployed BPEL process
- **-force**: Directs the compiler not to check the timestamps of the `.bpel`, `.wsdl`, and `.xml` files

We can use the following command to generate the BPEL process JAR archive with the revision number 1.0 for our travel example:

```
> bpelc -rev 1.0
```

This command generates the `bpel_TravelProcessCh4_1.0.jar` archive file, as shown:



```
Developer Prompt
C:\Prg\orabpel\Chapter4\TravelProcess>bpe1c -rev 1.0
-----
Oracle BPEL Processor Version 2.0
http://otn.oracle.com/bpel
Copyright (c) 2002-2004 - Oracle
(type bpe1c -help for help)
-----

bpe1c> validating "C:\Prg\orabpel\Chapter4\TravelProcess\Travel.bpel" ...
bpe1c> BPEL suitcase generated in: C:\Prg\orabpel\Chapter4\TravelProcess\bpe1_TravelProcessCh4_1.0.jar

bpe1c completed successfully.

C:\Prg\orabpel\Chapter4\TravelProcess>
```

The generated archive includes the BPEL source and the related WSDL and XML files. It also includes the process model file (in our example called `TravelModel.xml`) which is a normalized BPEL representation with an added `id` for each activity.

Deployment and Domains

We have several options to deploy our travel process:

- Copy the BPEL archive to the server domain manually
- Use `bpe1c` to deploy the process
- Use the `obant` utility to do the deployment
- Use the BPEL Console to deploy the BPEL process archive

Before we deploy, let's discuss the Oracle BPEL Server architecture. Each Oracle BPEL Server installation can be logically partitioned into several domains. The default domain is created automatically by the installation (called `default`). Additional domains can be created using BPEL Console; this is discussed later in the chapter. The domains are located in the `c:\orabpel\domains` directory.

To manually deploy a BPEL process we simply copy the JAR archive to the corresponding directory. In our case this is `c:\orabpel\domains\default\deploy`. The BPEL Server will automatically pick up the process.

Using `bpe1c` to deploy the process requires us to use the `-deploy <domain_id>` option, which directs the compiler to automatically deploy the archive to the specified domain.

The domain to which the deployment is done must be accessible via the file system. To deploy our travel example using `bpe1c` we need to use the following command:

```
> bpe1c -rev 1.0 -deploy default
```

The following screenshot shows the output:

```
Developer Prompt
C:\Prg\orabpel\Chapter4\TravelProcess>bpelc -rev 1.0 -deploy default
-----
Oracle BPEL Processor Version 2.0
http://otn.oracle.com/bpel
Copyright (c) 2002-2004 - Oracle
(type bpelc -help for help)
-----

bpelc> validating "C:\Prg\orabpel\Chapter4\TravelProcess\Travel.bpel" ...
bpelc> BPEL suitcase deployed to: C:\prg\orabpel\domains\default\deploy
bpelc completed successfully.

C:\Prg\orabpel\Chapter4\TravelProcess>
```

Ant Utility

The Oracle BPEL Process Manager provides the Ant utility called **obant**. This can be used to configure complex compilation and deployment scenarios. **obant** is just a wrapper around standard Ant, which sets the environment and then invokes the standard Ant Java task. To use it we have to prepare the corresponding project file, usually called **build.xml**. The project file for our travel example process is shown below:

```
<?xml version="1.0"?>
<project name="TravelProcessCh4" default="main" basedir=".">
    <property name="deploy" value="default"/>
    <property name="rev" value="1.0"/>
    <target name="main">
        <bpelc home="${home}" rev="${rev}" deploy="${deploy}"/>
    </target>
</project>
```

For more information on Ant, visit <http://ant.apache.org/>.

To compile and deploy our BPEL process we simply start **obant** from the command line. The output is shown in the following screenshot:

```
Developer Prompt
C:\Prg\orabpel\Chapter4\TravelProcess>obant
C:\Prg\orabpel\Chapter4\TravelProcess>SETLOCAL
Buildfile: build.xml

main:
  [bpelc] bpelc> validating "C:\Prg\orabpel\Chapter4\TravelProcess\Travel.bpel" ...
  [bpelc] bpelc> BPEL suitcase deployed to: C:\prg\orabpel\domains\default\deploy

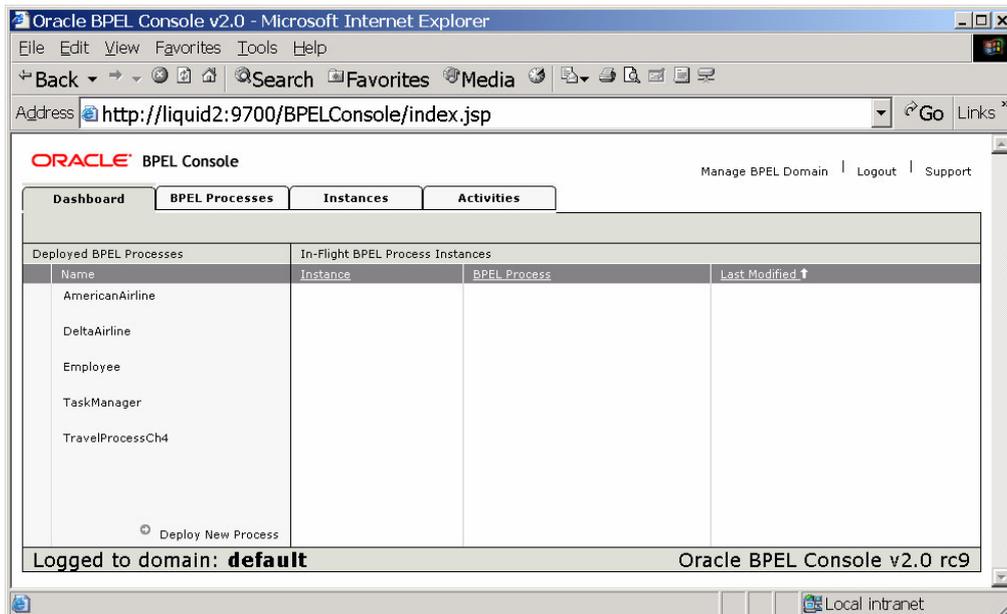
BUILD SUCCESSFUL
Total time: 13 seconds

C:\Prg\orabpel\Chapter4\TravelProcess>ENDLOCAL
C:\Prg\orabpel\Chapter4\TravelProcess>
```

Process Management with BPEL Console

Now that we have successfully deployed a BPEL process on the Oracle BPEL Server, let's execute it. In Chapter 2 we mentioned that each BPEL process is a web service. Therefore, to start the BPEL process we need to invoke it just like any other web service. This requires writing a web services client based on the WSDL. Because web services are not bound to a particular platform or programming language, we can do this using most languages (Java, C#, VB.NET, Delphi, etc.), applications (SAP, Navision, even Microsoft Office), tools (XML Spy), or other BPEL processes.

In addition to these options, Oracle BPEL Process Manager provides a BPEL Console through which we can execute, monitor, manage, and debug BPEL processes on a BPEL Server domain. The BPEL Console is accessible at <http://localhost:9700/BPELConsole/>. Of course we can replace `localhost` with the valid computer name URL. Once we enter the domain password we can start our travel process and create a new process instance by clicking the process name (`TravelProcessCh4`) on the BPEL Console dashboard, as shown in the following screenshot:



Note that in addition to the Travel Process, the Employee, American Airline, and Delta Airline web services have to be deployed as well. After clicking `TravelProcessCh4` we have to enter the input XML message (the default from the process descriptor is shown) and click the `Post XML Message` button:

Oracle BPEL Console v2.0 - Microsoft Internet Explorer

Address: <http://localhost:9700/BPELConsole/displayProcess.jsp?processId=TravelProcessCh4&revisionTag=1.0>

ORACLE BPEL Console Manage BPEL Domain | Logout | Support

Dashboard | **BPEL Processes** | Instances | Activities

BPEL Process: TravelProcessCh4 Version: 1.0 Lifecycle: Active

Statistics: [0 Open Instances](#) | [1 Complete Instances](#)

Testing this BPEL Process

About this BPEL Process
The Business Travel Process example.

Initiating a test instance

To create a new 'test' instance of this BPEL Process, fill the following text area with the XML representation of the input message and click on the 'Post XML Message' button.

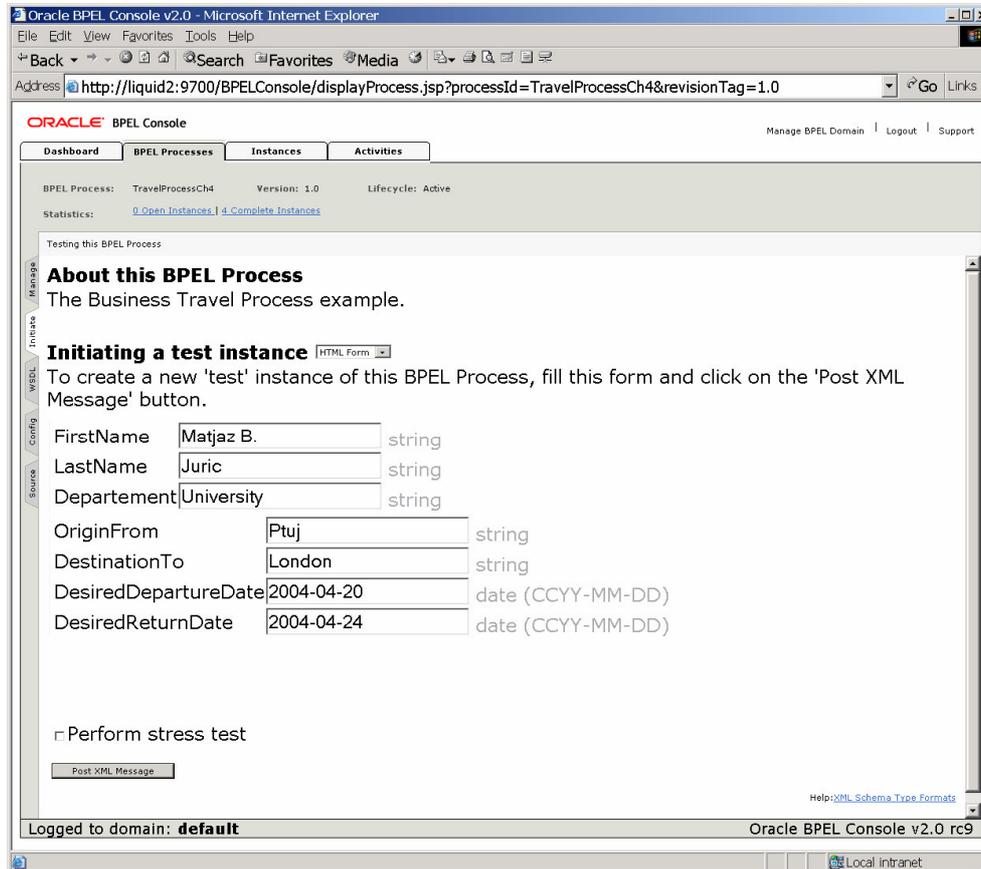
```
<TravelRequest xmlns="http://packtpub.com/bpel/travel/">
  <employee xmlns="http://packtpub.com/service/employee/">
    <FirstName>Matjaz B.</FirstName>
    <LastName>Juric</LastName>
    <Departement>University</Departement>
  </employee>
  <flightData xmlns="http://packtpub.com/service/airline/">
    <OriginFrom>Ptuj</OriginFrom>
    <DestinationTo>London</DestinationTo>
    <DesiredDepartureDate>2004-04-20</DesiredDepartureDate>
    <DesiredReturnDate>2004-04-24</DesiredReturnDate>
  </flightData>
</TravelRequest>
```

Perform stress test

Help: [XML Schema Type Formats](#)

Logged to domain: **default** Oracle BPEL Console v2.0 rc9

We can also switch to the HTML form and enter the necessary fields:



We now get a screen notifying us that the process instance is being processed asynchronously, because this is an asynchronous process. If this was a synchronous process, we would see the final result (returned through the `<reply>` activity) immediately.

Visual Flow

In the next step we can select the visual flow of the execution, instance auditing, or instance debugging. The visual flow of the instance graphically shows the execution of a BPEL process instance. We can monitor the execution of the process and its state (running, completed, canceled, or stale):

Oracle BPEL Console v2.0 - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://liquid2:9700/BPELConsole/displayInstance.jsp?referenceId=11d1def534ea1be0:1649b44:fe Go Links

ORACLE BPEL Console Manage BPEL Domain | Logout | Support

Dashboard BPEL Processes **Instances** Activities

Title: Instance #101 of TravelProcessCh4
 Reference Id: bpel://localhost/default/TravelProcessCh4~1.0/101
 BPEL Process: [TravelProcessCh4 \(v. 1.0\)](#)

Last Modified: 2004-08-19 23:36:30.058
 State: closed.completed
 Priority: 0

Visual representation of the history of this BPEL business flow [As of 8/19/04 11:38 PM] [Refresh View](#)

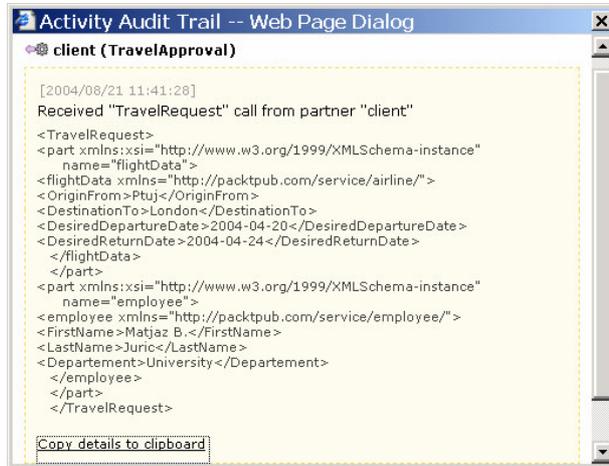
```

    graph TD
      start([start]) --> receive[receive  
client  
(TravelApproval)]
      receive --> subgraph RetrieveEmployeeTravelStatus
        assign1[assign  
Assign] --> invoke1[invoke  
employeeTrave...  
(EmployeeTrav...)]
        invoke1 --> assign2[assign  
Assign]
      end
      assign2 --> subgraph CheckFlightAvailability
        invoke2[invoke  
DeltaAirlines  
(FlightAvaila...)]
        invoke3[invoke  
AmericanAirlines  
(FlightAvaila...)]
      end
  
```

Logged to domain: **default** Oracle BPEL Console v2.0 rc9

Done. 43 entries rendered. Local intranet

The important thing is that we can click on each activity symbol (such as `<receive>`, `<assign>`, etc.) and we will see the corresponding XML input and output. This enables us to verify the processing of each activity. Clicking on the first `<receive>` activity (client `TravelApproval`) would open this screen, showing the received message, `TravelRequest`:



Instance Auditing

The audit view of the process instance, which we can activate by selecting the **Audit** tag, shows a complete BPEL process with the received and sent messages. This view is useful for auditing the messages exchanged by the process and the execution of other activities, particularly those manipulating data such as **<assign>**. The following screenshot shows the audit trail of our travel example process with the XML messages for each activity:

The screenshot shows the Oracle BPEL Console v2.0 interface in a Microsoft Internet Explorer browser. The address bar shows the URL: `http://liquid2:9700/BPELConsole/displayInstance.jsp?referenceId=11d1def534ea1be0:16`. The console displays the following information:

- Title:** Instance #301 of TravelProcessCh4
- Reference Id:** bpel://localhost/default/TravelProcessCh4~1.0/301
- BPEL Process:** TravelProcessCh4 (v. 1.0)
- Last Modified:** 2004-08-21 11:41:30.209
- State:** open.running
- Priority:** 0

The audit trail shows the following events:

- [2004/08/21 11:41:27]** New instance of BPEL process "TravelProcessCh4" initiated (# "301").
- client (TravelApproval)** [2004/08/21 11:41:28] Received "TravelRequest" call from partner "client". The XML message is:


```
<TravelRequest>
  <part xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" name="flightData">
    <flightData xmlns="http://packtpub.com/service/airline/">
      <OriginFrom>Ptuj</OriginFrom>
      <DestinationTo>London</DestinationTo>
      <DesiredDepartureDate>2004-04-20</DesiredDepartureDate>
      <DesiredReturnDate>2004-04-24</DesiredReturnDate>
    </flightData>
  </part>
  <part xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" name="employee">
    <employee xmlns="http://packtpub.com/service/employee/">
      <FirstName>Matjaz B.</FirstName>
      <LastName>Juric</LastName>
      <Department>University</Department>
    </employee>
  </part>
</TravelRequest>
```
- Assign** [2004/08/21 11:41:28] Updated variable "EmployeeTravelStatusRequest". The XML message is:


```
<EmployeeTravelStatusRequest>
  <part xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" name="employee">
    <employee xmlns="http://packtpub.com/service/employee/">
      <FirstName>Matjaz B.</FirstName>
      <LastName>Juric</LastName>
      <Department>University</Department>
    </employee>
  </part>
</EmployeeTravelStatusRequest>
```
- employeeTravelStatus (EmployeeTravelStatus)** [2004/08/21 11:41:29] Invoked 2-way operation "EmployeeTravelStatus" on partner "employeeTravelStatus". The XML messages are:


```
<EmployeeTravelStatusRequest>
  <part xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" name="employee">
    <employee xmlns="http://packtpub.com/service/employee/">
      <FirstName>Matjaz B.</FirstName>
      <LastName>Juric</LastName>
      <Department>University</Department>
    </employee>
  </part>
</EmployeeTravelStatusRequest>
<EmployeeTravelStatusResponse>
  <part xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" name="travelClass">
    <travelClass xmlns="http://packtpub.com/service/employee/">Economy</travelClass>
  </part>
</EmployeeTravelStatusResponse>
```

The status bar at the bottom indicates "Logged to domain: default" and "Oracle BPEL Console v2.0 rc9". The browser status bar shows "Done. 43 entries rendered." and "Local intranet".

Debugging

The debug view of a process instance (accessible via the **Debug** tag) shows the BPEL source code. Clicking on the underlined variable names provides access to variable content. We can debug already completed instances or instances that are still running; this is called in-flight debugging. The debug view shows the current state of the instance. If we use in-flight debugging, the point where execution is paused is shown highlighted. The following figure shows the debug view of our process instance, once completed:

Oracle BPEL Console v2.0 - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://liquid2:9700/BPELConsole/displayInstance.jsp?referenceId=11d1def534ea1be0:16

ORACLE BPEL Console Manage BPEL Domain | Logout | Support

Dashboard BPEL Processes Instances Activities

Title: Instance #301 of TravelProcessCh4
 Reference Id: bpel://localhost/default/TravelProcessCh4~1.0/301
 BPEL Process: TravelProcessCh4 (v. 1.0)
 Last Modified: 2004-08-21 11:41:40.113
 State: closed.completed
 Priority: 0

This instance is already completed. [Show Activity IDs](#)

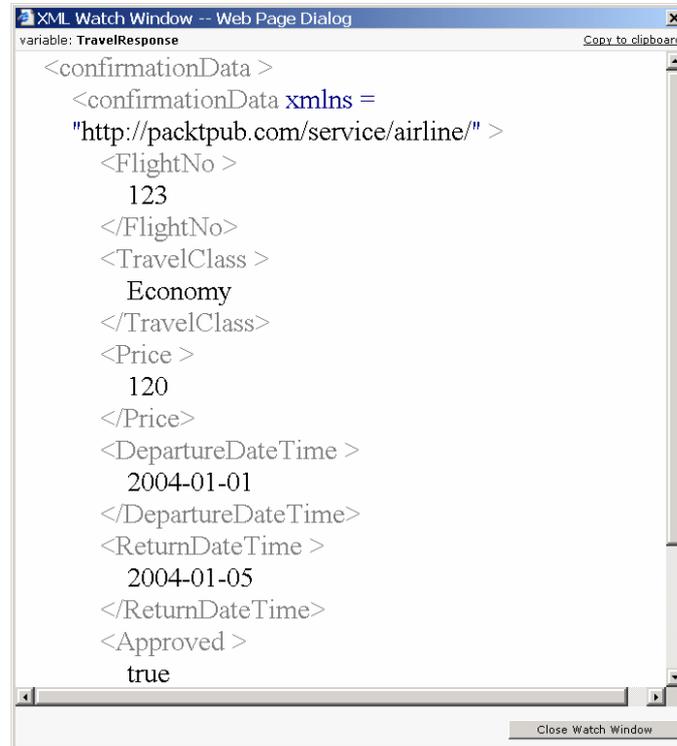
```

<!-- Asynchronous BPEL process -->
<process name="BusinessTravelProcess" targetNamespace="http://packtpub.com/bpel/travel/" >
  <partnerLinks>
    <partnerLink myRole="travelService" name="client" partnerLinkType="trv:travelLT"
      partnerRole="travelServiceCustomer" />
    <partnerLink name="employeeTravelStatus" partnerLinkType="emp:employeeLT"
      partnerRole="employeeTravelStatusService" />
    <partnerLink myRole="airlineCustomer" name="AmericanAirlines" partnerLinkType="ain:flightLT"
      partnerRole="airlineService" />
    <partnerLink myRole="airlineCustomer" name="DeltaAirlines" partnerLinkType="ain:flightLT"
      partnerRole="airlineService" />
  </partnerLinks>
  <variables>
    <!-- input for this process -->
    <variable name="TravelRequest" messageType="trv:TravelRequestMessage" />
    <!-- input for American and Delta web services -->
    <variable name="FlightDetails" messageType="ain:FlightTicketRequestMessage" />
    <!-- output from BPEL process -->
    <variable name="TravelResponse" messageType="ain:TravelResponseMessage" />
    <!-- fault to the BPEL client -->
    <variable name="TravelFault" messageType="trv:TravelFaultMessage" />
  </variables>
  <faultHandlers>
    <catchAll>
      <sequence>
        <!-- Create the TravelFault variable -->
        <assign>
          <copy>
            <from expression="string('Other fault')" />
            <to part="error" variable="TravelFault" />
          </copy>
        </assign>
        <invoke inputVariable="TravelFault" operation="ClientCallbackFault" partnerLink="client"
          portType="trv:ClientCallbackPT" />
      </sequence>
    </catchAll>
  </faultHandlers>
  <sequence>
    <!-- Receive the initial request for business travel from client -->
  
```

Logged to domain: **default** Oracle BPEL Console v2.0 rc9

Resizing View (11 of 10) Local Intranet

Clicking on the **TravelResponse** variable, for example, gives us the following output:



```
XML Watch Window -- Web Page Dialog
variable: TravelResponse
Copy to clipboard

<confirmationData >
  <confirmationData xmlns =
    "http://packtpub.com/service/airline/" >
    <FlightNo >
      123
    </FlightNo>
    <TravelClass >
      Economy
    </TravelClass>
    <Price >
      120
    </Price>
    <DepartureDateTime >
      2004-01-01
    </DepartureDateTime>
    <ReturnDateTime >
      2004-01-05
    </ReturnDateTime>
    <Approved >
      true
    </Approved >
  </confirmationData >
</confirmationData >
```

Overview of Other Console Functions

Using the **BPEL Processes** tab we can manage the process lifecycle and process state. The state of a process can be *on* or *off*. When the state of a process is off, new instances cannot be created and access to existing instances is blocked. The lifecycle of a process can be *active* or *retired*. When a process is retired, new instances cannot be created. Existing instances can, however, complete normally. We can also get configuration data, WSDL and endpoint locations, and a source view.

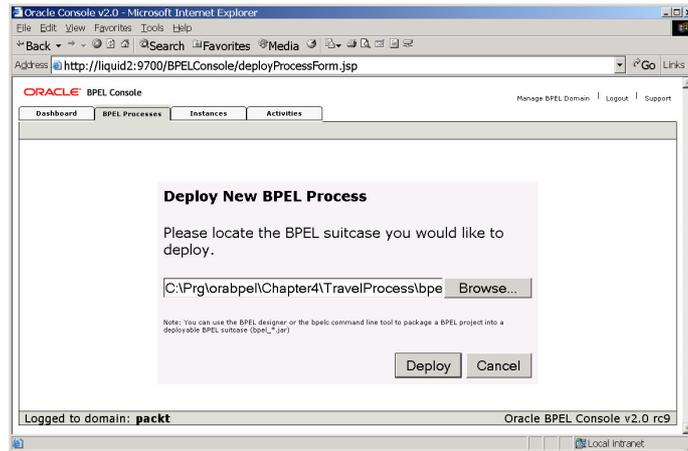
Under the **Instances** tab we can overview process instances. We can archive and purge instances, remove completed process instances, and supervise those that have not completed yet. A BPEL process instance can have the following states:

- Running
- Completed
- Canceled
- Stale

Under the **Activities** tab we can locate activities by name and find relations to instances and processes. An activity can also have four states: open, completed, canceled, or stale.

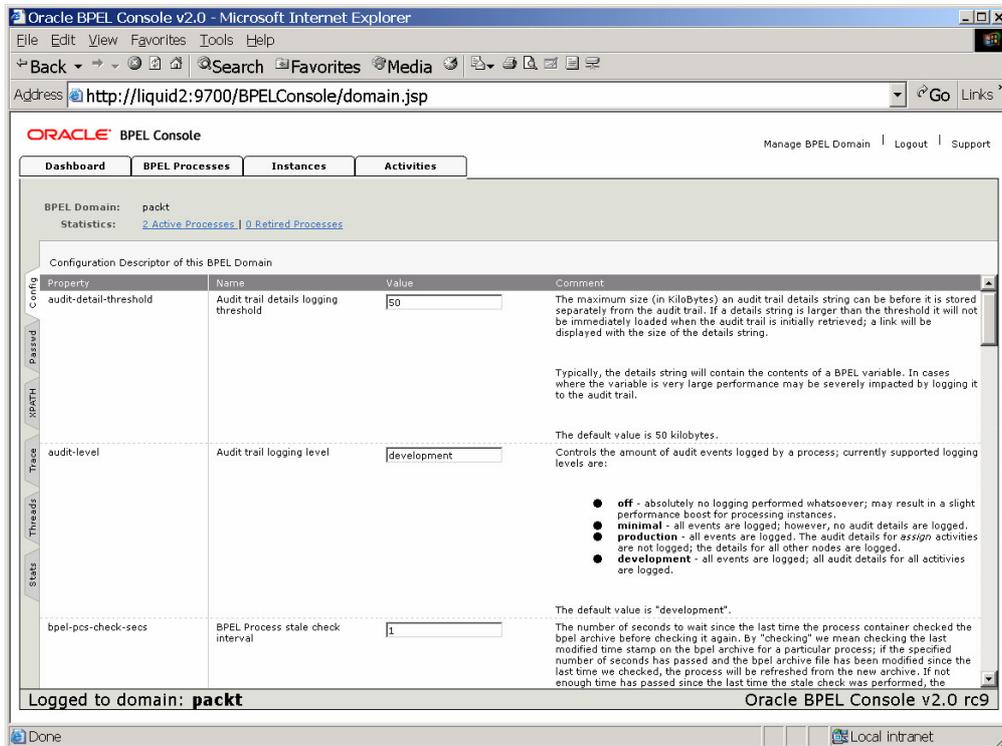
Deploying Processes

We have already mentioned that new processes can be deployed using BPEL Console. To do this we first have to generate the process JAR archive (using **bpelc** or **obant**). We then click the **Deploy New Processes** link on the **Dashboard** or **BPEL Processes** tabs. We then specify the full path to the process JAR archive and press the **Deploy** button, as shown on the screenshot below:



Management

BPEL Console also provides tools for BPEL domain management. These can be accessed by pressing the **Manage BPEL Domain** link in the upper right corner of the screen, which shows the following screen:



The management console has the following important options:

- Configuration descriptor
- Setting the password
- Setting the logging configuration (trace)
- Thread allocation statistic
- Runtime performance statistic
- List of XPath extension functions

The configuration descriptor enables us to set various important parameters that affect the BPEL Server operation. These parameters include:

- Process instance stale check interval (specified in seconds)
- Allocation of invocation threads and load factor
- Instance cache size (minimum and maximum)
- Persistence and database parameters
- Recovery agent settings
- Audit trail and other settings

We can also set a parameter that defines the behavior of the server when performing the `<assign>` activity. This parameter is called **Relax BPEL4WS1.1 spec assign rules**. BPEL specification sets certain rules by assignments (discussed in Chapter 3). For example, null assignments are not allowed by default. If these rules are too restrictive for us, we can change the behavior by setting this parameter to **true**. However, doing this is not recommended because it can hinder portability of BPEL processes. The default value of this parameter is **false**.

Performance Tuning

The above-mentioned parameters on stale check, threads and load factor, cache size, etc. affect the performance of the Oracle BPEL Server. Together with the runtime performance statistic (**Stats** tab) and thread allocation statistic (**Thread** tab), they can be used to tune the performance. The runtime performance statistics provide comprehensive data about the execution time of processes and a breakdown of times by activities, as shown in the following screenshot:

The screenshot shows the Oracle BPEL Console v2.0 interface. The main content area displays runtime statistics for the 'default' BPEL Domain. It is divided into three sections: Synchronous BPEL process statistics, Asynchronous BPEL process statistics, and Request breakdown. Each section shows a table of activities with their respective average execution times and ranges (min/max).

Category	Activity	Stats	Average
Synchronous BPEL process statistics	Employee 1.0	(2 stats, min = 200 ms, max = 310 ms)	255.0 ms
	Asynchronous BPEL process statistics		
Asynchronous BPEL process statistics	TravelProcessChk 1.0	(1 stat, min = 8782 ms, max = 8782 ms)	9782.0 ms
	AmericanAirline 1.0	(2 stats, min = 390 ms, max = 1672 ms)	1031.0 ms
	DeltaAirline 1.0	(2 stats, min = 230 ms, max = 2603 ms)	1416.5 ms
Request breakdown	eng-composite-request	(15 stats, min = 100 ms, max = 2603 ms)	616.13 ms
	eng-single-request	(43 stats, min = 0 ms, max = 1552 ms)	146.27 ms
	eng-callback	(2 stats, min = 10 ms, max = 10 ms)	10.0 ms
	eng-finalize	(2 stats, min = 0 ms, max = 10 ms)	5.0 ms
	eng-until	(2 stats, min = 0 ms, max = 10 ms)	0.0 ms
	eng-manage	(2 stats, min = 0 ms, max = 10 ms)	5.0 ms
	load-workitem	(2 stats, min = 10 ms, max = 30 ms)	20.0 ms
	load-wi-datasource	(2 stats, min = 10 ms, max = 30 ms)	20.0 ms
	wi-load	(2 stats, min = 10 ms, max = 30 ms)	20.0 ms
	populate-context	(6 stats, min = 0 ms, max = 0 ms)	0.0 ms
	handle-workitem	(28 stats, min = 0 ms, max = 1552 ms)	156.64 ms
	do-perform	(28 stats, min = 0 ms, max = 1552 ms)	156.28 ms
	check-schedulable	(28 stats, min = 0 ms, max = 0 ms)	0.0 ms
	check-expirable	(28 stats, min = 0 ms, max = 0 ms)	0.0 ms
	actual-perform	(28 stats, min = 0 ms, max = 1552 ms)	152.35 ms
	employeeTravelStatus	(2 stats, min = 0 ms, max = 0 ms)	0.0 ms
	Assign	(9 stats, min = 0 ms, max = 60 ms)	11.11 ms
	parse	(2 stats, min = 0 ms, max = 0 ms)	0.0 ms

At the bottom of the console, it shows 'Logged to domain: default' and 'Oracle BPEL Console v2.0 rc9'.

The thread allocation statistics provide information on the usage of threads and their allocation, and on the number of requests on BPEL processes, as shown in the next screenshot:

The screenshot displays the Oracle BPEL Console v2.0 interface in a Microsoft Internet Explorer browser window. The address bar shows the URL: `http://liquid2:9700/BPELConsole/domain.jsp?mode=thread`. The page title is "ORACLE BPEL Console".

Navigation tabs include: Dashboard, BPEL Processes, Instances, and Activities. The current view is "Thread Allocation Statistics for this BPEL Domain".

Statistics: [5 Active Processes](#) | [0 Retired Processes](#)

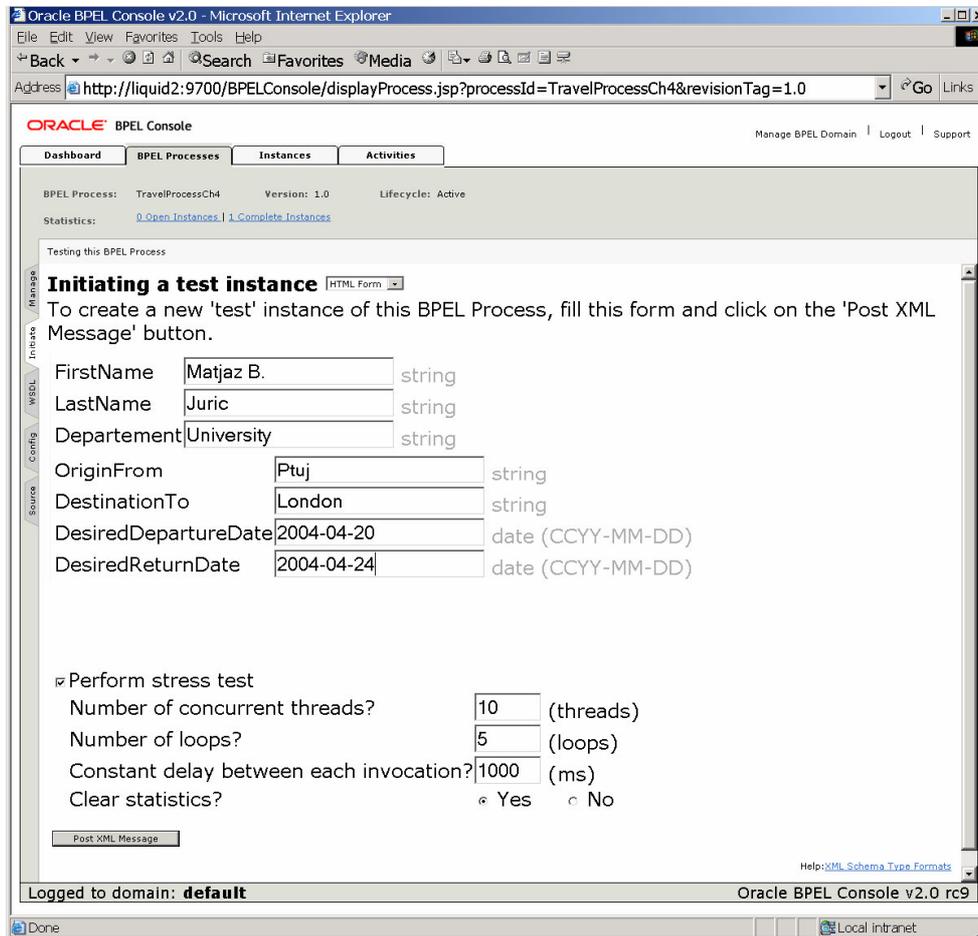
Thread Allocation Statistics for this BPEL Domain

Pending Requests	Scheduled	Active
BPEL Domain Management Requests	0	0
BPEL Process Management Requests	0	0
Transaction Coordination Requests	0	0
Callback Requests	0	0
Activity Execution Requests	0	0
New Instance Requests	0	0

Thread Allocation Activity	
Active threads	0
Active invocation threads	0
Active engine threads	0
Highest number of active threads	3
Total number of allocated threads allocated over time	12
Average JMS thread allocation overhead	605 (ms)
Average lifetime for allocated threads	0 (ms)
Average number of messages processed per thread	0
Load Factor (# of scheduled messages/ # of working messages)	n/a

Logged to domain: **default** Oracle BPEL Console v2.0 rc9

When creating a BPEL process test instance the Oracle BPEL Server provides an option through which we can perform stress tests. Stress tests enable us to monitor the performance and to do load testing of processes. With the performance statistic we can identify the possible bottlenecks and optimize the performance. To perform a stress test we simply select the **Perform stress test** option as shown in the following screenshot. We then have to specify the number of concurrent threads allocated to the process, number of loops for the test, and the delay between invocations. We also have to select whether to clear statistics before running the stress test. This way we can identify the most appropriate number of threads:



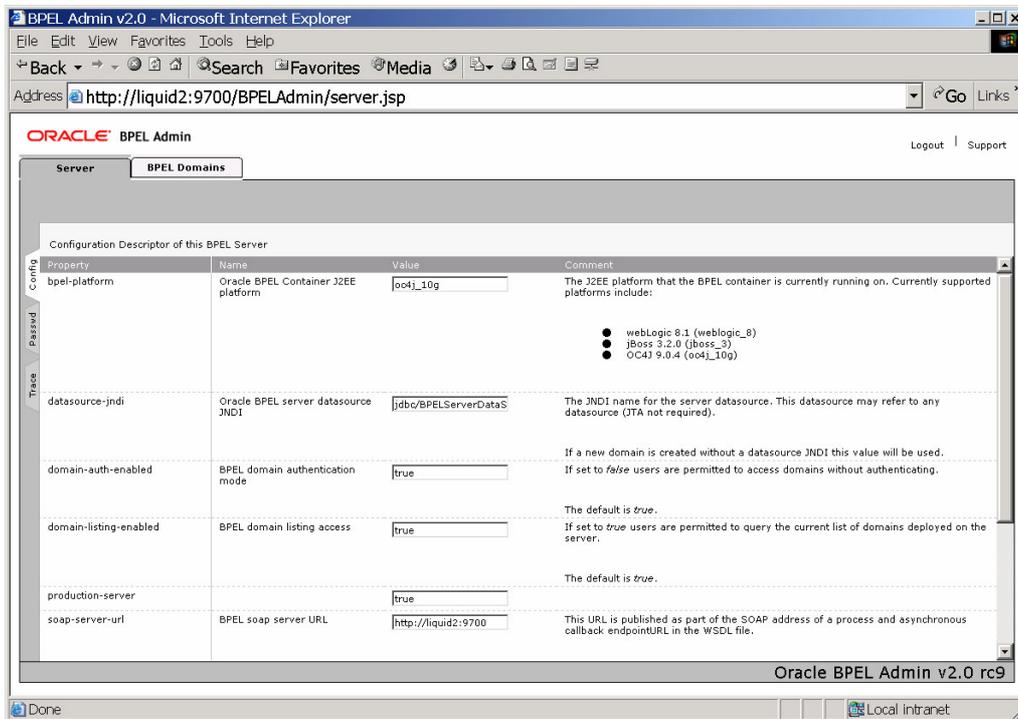
Note that the default download edition of Oracle BPEL Process Manager on Windows platform bundles Oracle Lite as the database, which will not yield meaningful results for a stress test. The BPEL Server should be configured to use a production-quality database (Oracle, SQL Server, or DB2) before doing stress testing. A technical note on <http://otn.oracle.com/bpel> describes how to configure the Oracle BPEL Process Manager for a database server other than Oracle Lite.

Domains and Administration

BPEL Console can also be used for server administration. The BPEL administration can be accessed if we follow the [Goto BPEL Admin](#) link on the main logon page. We have to supply the administration password, which is initially **oracle**. The BPEL Admin has two major functions:

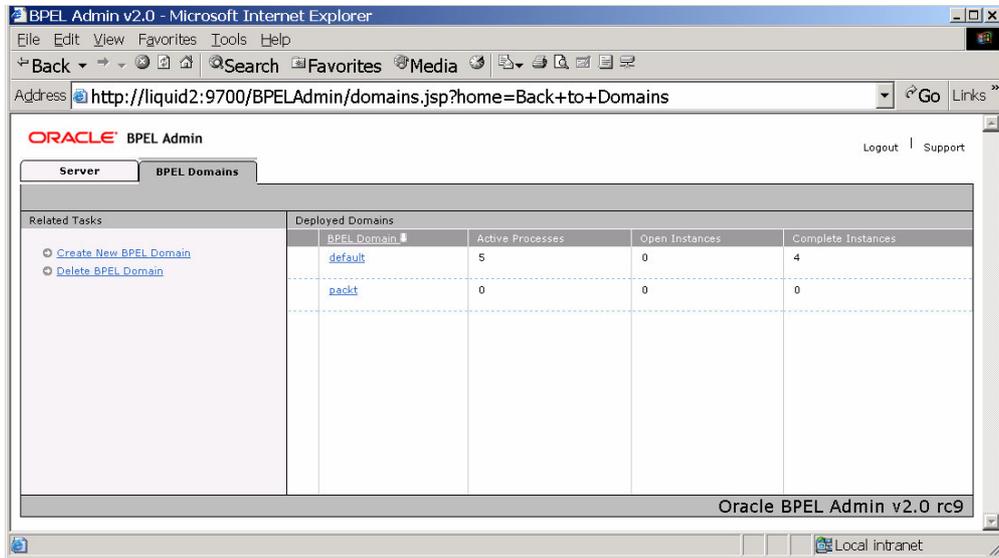
- Administration of server-related parameters and administration passwords
- Managing BPEL domains

The administration of server-related parameters includes three tabs: **Config**, **Password**, and **Trace**:

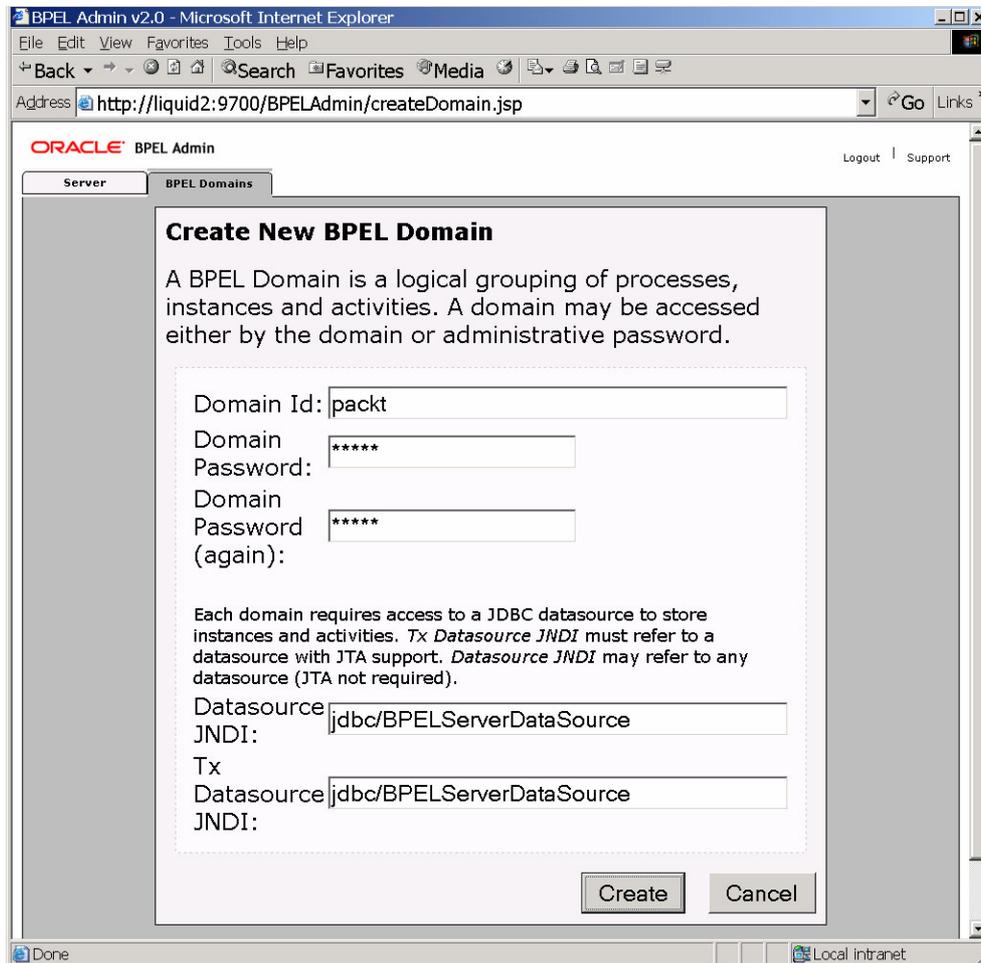


Within **Config** we can modify various server parameters, including J2EE container platform (OC4J, JBoss, WebLogic), data sources, and BPEL SOAP server URL. We can also specify whether users need to supply a password when they log on to a BPEL domain and whether they can see a list of available domains.

This brings us to the BPEL domains. We have already mentioned that Oracle BPEL Server is organized into domains. Since domains enable us to logically organize business processes, using several domains is recommended in real-world scenarios. After installation, the **default** domain is created automatically. Additional domains can be created under the **BPEL Domains** tab, as shown in the following screenshot:



If we follow the [Create New BPEL Domain](#) link we can create new domains. We have to specify the domain ID, password, and JNDI (Java Naming and Directory Interface) addresses for regular and transactional data sources.:



With this we have concluded our review of the BPEL Console. In the next section we look at the BPEL Designer.

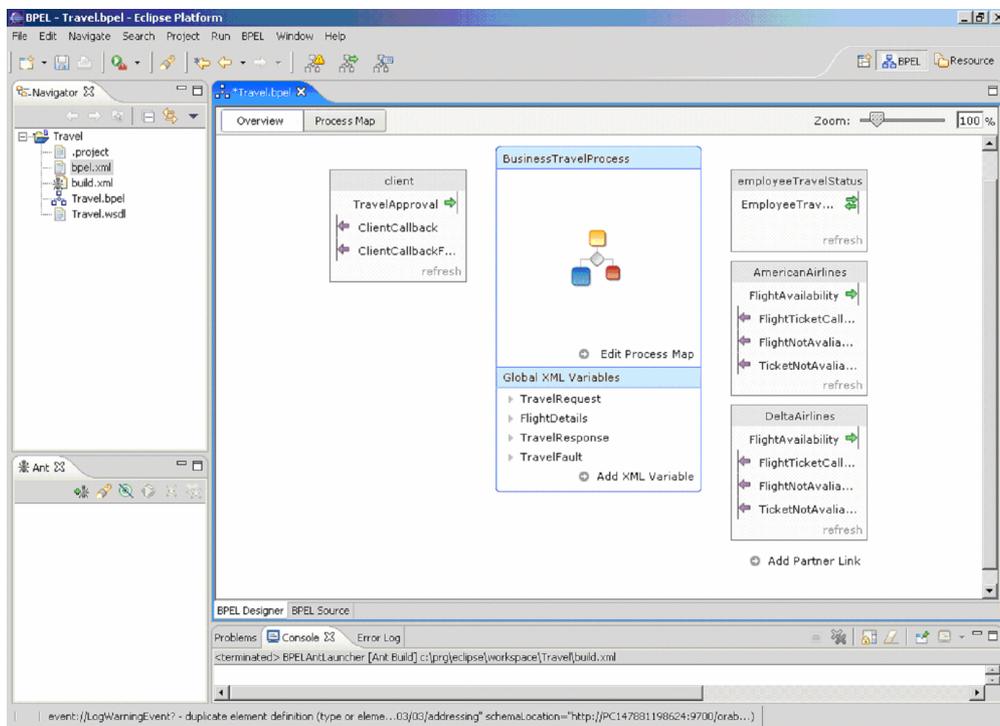
Graphical Development with BPEL Designer

Writing BPEL processes by hand as we have done in previous chapters can become time consuming. Therefore Oracle has developed the BPEL Designer, which enables graphical development of BPEL processes. Instead of writing BPEL code we can develop processes in a graphical environment where we can add activities using drag-and-drop. BPEL Designer simplifies development and makes it faster. In addition to drag-and-drop modeling it provides a browser through which we can locate web services. It also provides a copy assistant, an XPath editor, and the ability to compile and deploy a process on the BPEL Server with one mouse click.

Oracle BPEL Designer natively supports BPEL version 1.1, so we can also use it for developing BPEL processes that will be deployed on other BPEL servers. BPEL Designer supports Oracle-specific functions such as user tasks and Java embedding. However, using these limits portability.

The BPEL Designer has been developed as an Eclipse plug-in. Therefore we should get familiar with the basics of the Eclipse platform (<http://www.eclipse.org/>). BPEL Designer 1.0 requires that we install Eclipse 3.0. For detailed installation instructions please refer to Oracle tutorials, which can be downloaded from <http://www.oracle.com/technology/products/ias/bpel/index.html>.

The following screenshot shows the main BPEL Designer screen with the opened Travel process (the example used in this and previous chapters). It shows the overview of the process with partner links and global XML variables exposed:



Here we can add partner links and variables to our process.

Partner Links and Web Services

To add a partner link to the process, we simply click **Add Partner Link**, located in the lower right corner of the main window. After entering the partner link name and WSDL location, the designer will help us in selecting the partner link types and roles. This is shown in the following screenshot:

Oracle BPEL Designer -- Web Page Dialog

New partnerLink Wizard

Use this form to configure the attributes of the element to be created

name
AmericanAirline
loading...

wsdlLocation
http://localhost:9700/orabpel/default/AmericanAirline/AmericanAirline?wsdl refresh
loading...

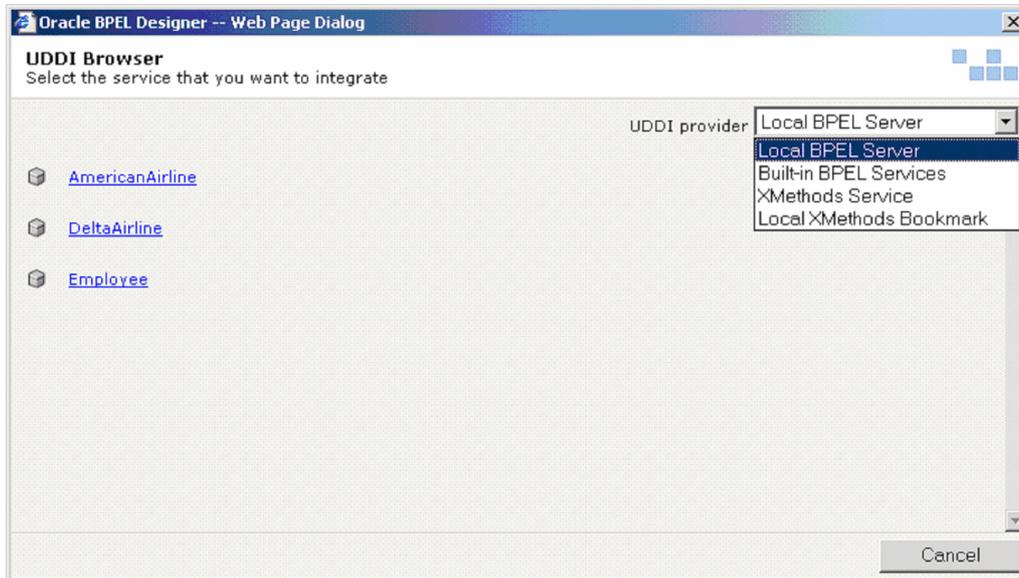
partnerLinkType
aln:flightLT
loading...

partnerRole
airlineService
loading...

myRole
airlineCustomer
loading...

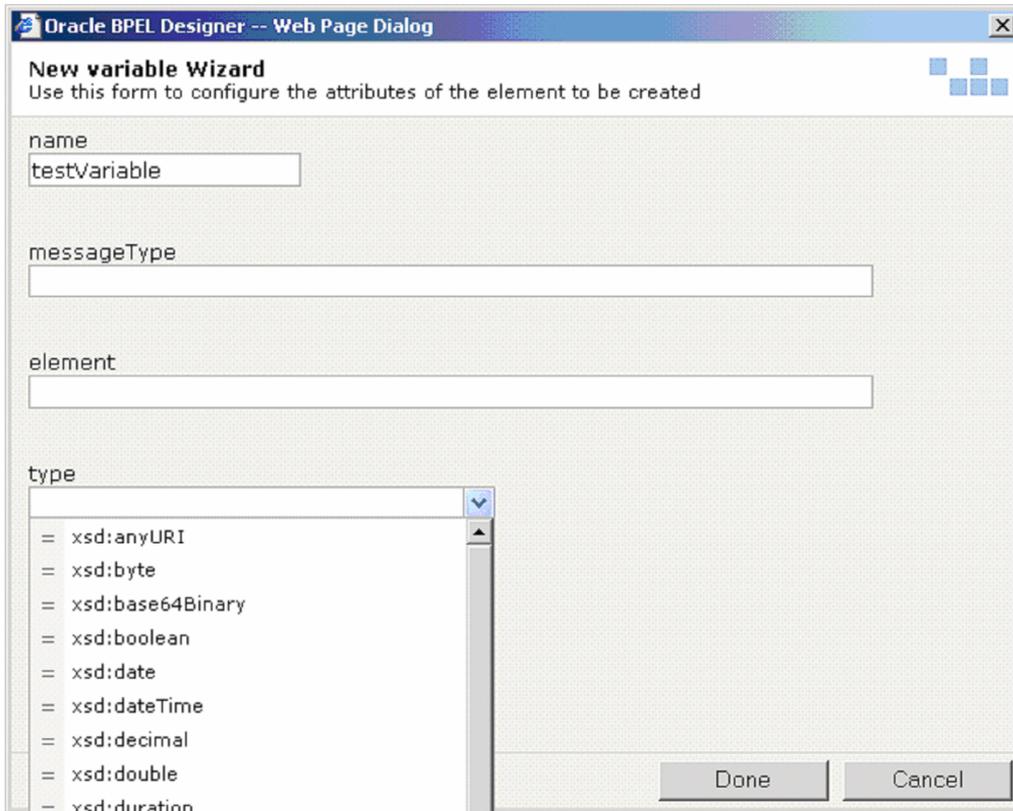
Done Cancel

If we do not know the exact location of the WSDL, we can use the UDDI browser, through which we can locate and select the appropriate web service. The web service can be located on the local computer, or we can use a UDDI registry. Later we will discuss BPEL Server's built-in services. The following screenshot shows the view on the local services:



Variables

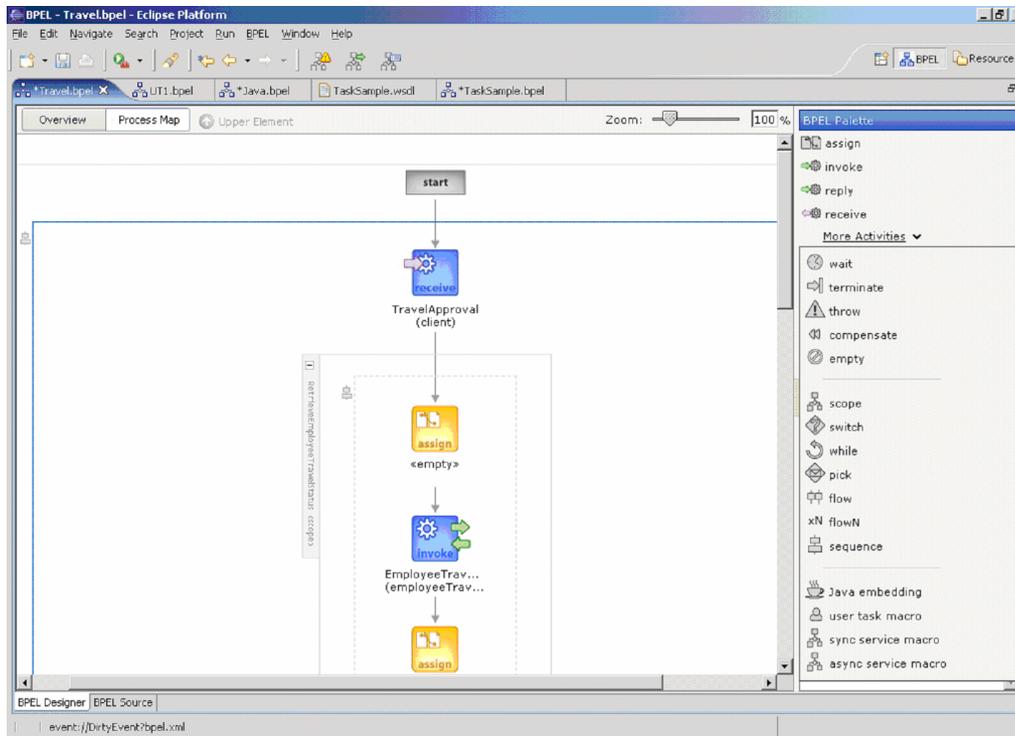
By following the [Add XML Variable](#) link, we can add a global variable to the process. Variables in BPEL processes can be defined globally or within scopes. Adding them with BPEL Designer requires us to fill out the following form:



We have to enter the variable name and type, which can be a message type, an element, or an XML Schema type.

Process Map

Let us now switch from the [Overview](#) to the [Process Map](#) view. BPEL Designer will show the graphical representation of the process, similar to what we have seen in the BPEL Console. In this view we can click on each activity to get the details (in the right-hand window). We can also add activities by dragging-and-dropping them on the required location in the process. As shown in the following figure, all standard BPEL and Oracle-specific activities are supported:



We can also right-click on each activity and select the required option from the menu. If we select an **<assign>** activity, we can select the **Add Copy Rule** option. This opens the Copy Rule window where we can enter the details of a copy activity:

Oracle BPEL Copy Customizer -- Web Page Dialog

Copy Rule
Use this form to customize this copy rule

From Variable
XPath Query

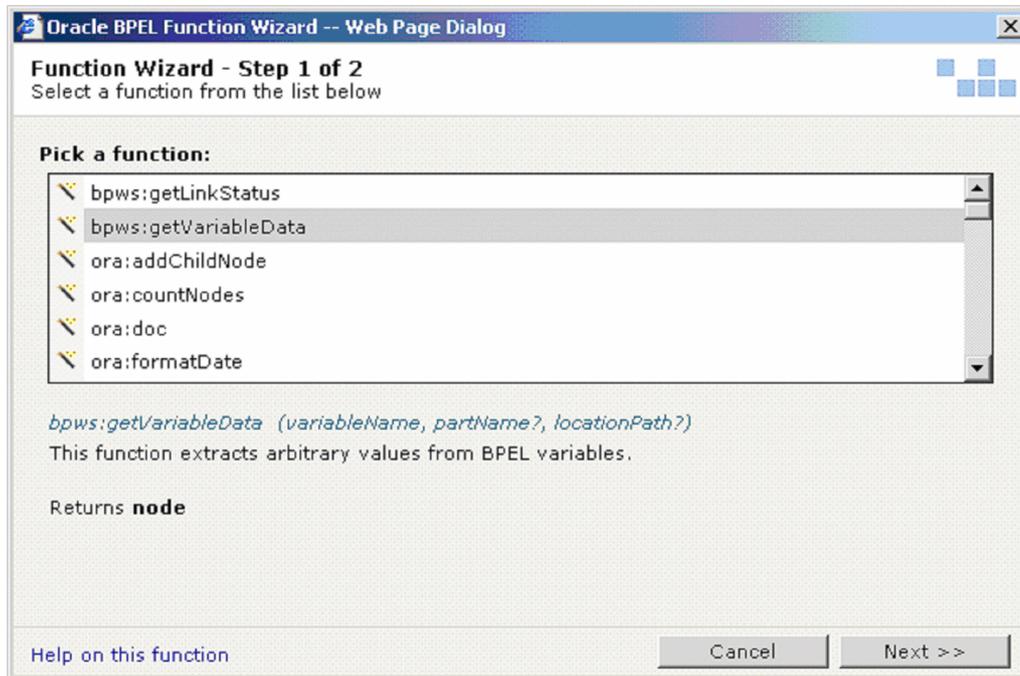
Expression

Literal

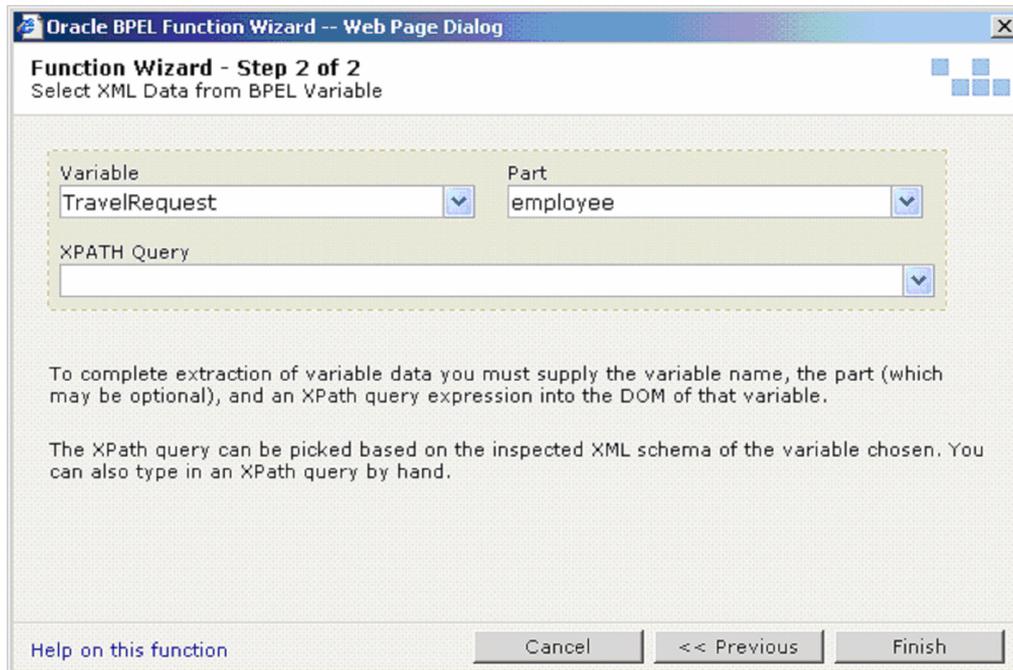
To Variable
XPath Query

Done Cancel

To enter an expression we can open the BPEL Function Wizard. The wizard helps us to compose an XPath expression. In the first step we have to select a function from the list of available functions. Note that it also offers some Oracle-specific functions (those with **ora** prefix), which are discussed later in this chapter:



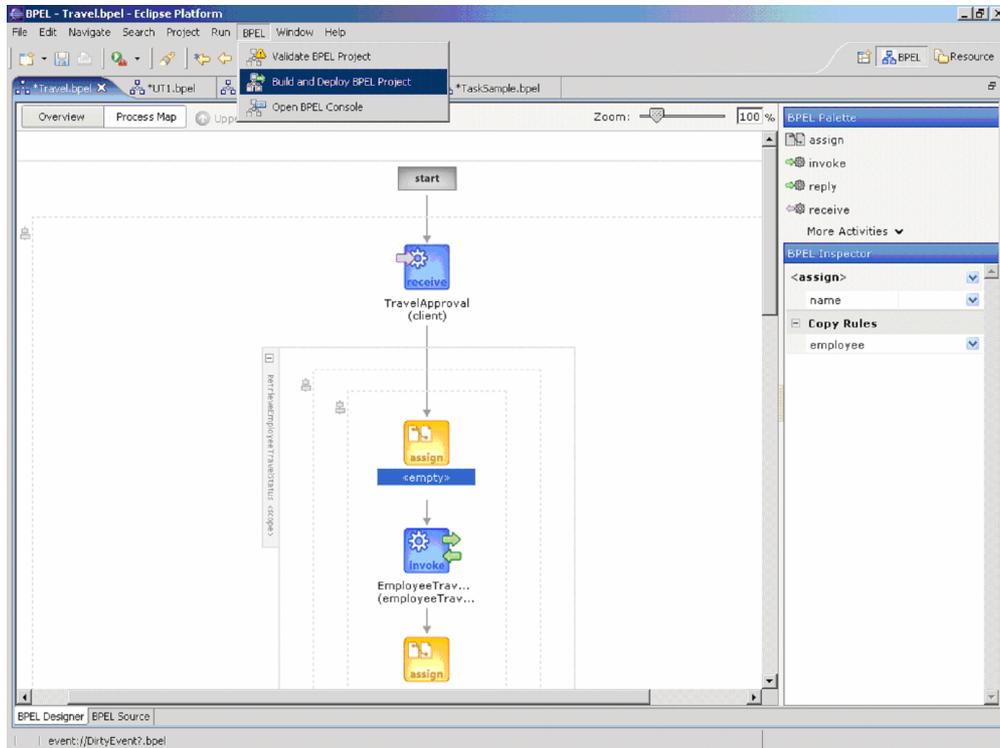
After selecting the function, we have to fill in the required parameters. In our case we have selected the `bpws:getVariableData()` function. Therefore we have to enter the variable, part, and XPath query (optional):



BPEL Designer also provides the source-code view where we can edit the BPEL code directly. Changes made in source view are reflected immediately in the BPEL Designer visual representation, and vice versa.

Building and Deploying

BPEL Designer offers direct compilation and deployment on the Oracle BPEL Server. This can be done from the toolbar or from the BPEL menu. In addition to building and deploying, we can also validate our project and open the BPEL Console:



For more information about the BPEL Designer refer to Oracle documentation. In the next section we look at the Oracle-specific functions of the BPEL Process Manager. We will start with the Oracle-specific extension functions.

Oracle-specific Functions

In Chapters 2 and 3 we saw that BPEL is very flexible with respect to the expression and query language. By default we use XPath 1.0; however, we can use any other language supported by the BPEL server. The idea behind this flexibility has been to open up BPEL for future versions of XPath (and XQuery).

XPath 1.0 does not provide all functions necessary to develop BPEL processes. Therefore, the BPEL specification defines additional functions such as `getVariableData()`, `getVariableProperty()`, and `getLinkStatus()`. Oracle BPEL Process Manager provides additional extension functions to simplify the development.

Using these functions limits the portability of BPEL processes, because these functions will not be available in other BPEL servers.

Oracle extension functions are defined in the following namespace URI:
<http://schemas.oracle.com/xpath/extension>. We will use the `ora` prefix for this namespace,
which corresponds to the following XML declaration
`xmlns:ora="http://schemas.oracle.com/xpath/extension"`.

The extension functions are related to:

- Transformation and query support
- Data and array manipulation
- XML manipulation
- Date and time expressions
- Process identification

All Oracle-specific functions can be accessed using BPEL Designer's Function Wizard.

Transformation and Query Support

In real-world business processes we often have to match the schema of our XML document to the schema required by the partner web service. Consider our travel process example. Here we designed both the process and the partner web services, so we only had to perform minimal transformations for calling the Employee or Airlines web services. In real-world examples this will often not be the case and we will have to make more complex transformations.

To perform the transformations, we can use the BPEL `<assign>` activity. As this can be time consuming, Oracle provides an XSLT engine and an extension function through which we can activate the XSLT engine. This enables us to use XSLT to do more complex data transformations. Using XSLT is more appropriate than using `<assign>` because XSLT is the standard transformation language for XML. Also, sometimes we already have the stylesheets for transformation. This way we can easily integrate them into BPEL processes.

To activate the XSLT engine we use the `ora:processXSLT()` function. The function requires two parameters, the XSLT stylesheet and the XML input on which the transformation should be made. The result of the function is the transformed XML. The syntax is:

```
ora:processXSLT('stylesheet', 'XML_input')
```

Usually we use this function within the `<assign>` activity, in the `<from>` clause. For example, to modify our travel process and make a more complex transformation to prepare the input for the Employee web service, we could use the XSLT engine, as shown in the following code excerpt:

```
<assign>
  <copy>
    <from expression="ora:processXSLT('employee.xslt',
      bpws:getVariableData('TravelRequest', 'employee') )"/>
    <to variable="EmployeeTravelStatusRequest" part="employee"/>
  </copy>
</assign>
```

For this code to work we must create the `employee.xslt` stylesheet and deploy it with the process. For more information on XSLT please refer to <http://www.w3.org/TR/xslt>.

In addition to the XSLT engine, Oracle BPEL Process Manager also provides:

- An XQuery engine
- An XSQL engine

With the XQuery engine we can perform complex queries on XML documents, going beyond the capabilities of XPath. We can use the built-in XQuery engine through the `ora:processXQuery()` function. We have to provide the query template and the context XML on which the query should be performed:

```
ora:processXQuery('query_template', 'XML_context')
```

We will use the function from the `<assign>` activity. Suppose we would like to create the `EmployeeTravelStatusResponse` with an XQuery. We would have to create the query and store it into the `query.xq` file and use the following code snippet:

```
<assign>
<copy>
  <from expression="ora:processXQuery(query.xq',
                                bpws:getVariableData('EmployeeTravelStatusRequest',
                                'employee'))"/>
  <to variable="EmployeeTravelStatusResponse" part="employee"/>
</copy>
</assign>
```

To process only a specific item, we can use the `ora:processXQueryItem()` function. The syntax is similar to `ora:processXQuery()`; here we have to provide the item:

```
ora:processXQueryItem('query_template', 'item', 'XML_context')
```

For more information on XQuery please refer to <http://www.w3.org/XML/Query>.

In a similar way we can use the Oracle XSQL engine. It can be activated using the `ora:processXSQL()` function. We have to provide the XSQL template and the input XML on which the query should be performed:

```
ora:processXSQL('query_template', 'XML_input')
```

Data and Array Manipulation

Data manipulation in BPEL is done within the `<assign>` activity, where we can use XPath and BPEL functions in the `<from>` and `<to>` clauses. In addition, Oracle provides several custom functions that ease data manipulation considerably.

A very important aspect in data manipulation is arrays. In Chapter 3 we mentioned that arrays in BPEL are realized with XML elements, which can occur more than once. In XML schema they are identified with the `maxOccurs` attribute, which can be set to a specific value or can be unbounded (`maxOccurs="unbounded"`). The items are addressed with the XPath `position()` function, as shown in the following example:

```
<assign>
<copy>
```

```

    <from variable="TicketOffer"
        part="ticket"
        query="/item[position()=1]"/>
    <to variable="FirstOffer" part="ticket"/>
</copy>
</assign>

```

The short notation is:

```

<assign>
  <copy>
    <from variable="TicketOffer"
        part="ticket"
        query="/item[1]"/>
    <to variable="FirstOffer" part="ticket"/>
  </copy>
</assign>

```

Often we need to dynamically address the items. Instead of hard-coding the index we can use a variable, such as:

```

<variable name="position" type="xsd:integer"/>

```

We could then create the XPath query expression, store it in a variable, and then use this variable to address the desired item, as shown in the following example:

```

<assign>
  <copy>
    <from expression="concat('/item[' ,
        bpws:getVariableData('position'), '']")"/>
    <to variable="itemAddress"/>
  </copy>
  <copy>
    <from expression="bpws:getVariableData('TicketOffer', 'ticket',
        bpws:getVariableData('itemAddress'))"/>
    <to variable="SelectedOffer" part="ticket"/>
  </copy>
</assign>

```

Alternatively we can use an Oracle-specific function called `ora:getElement()`. The function takes four parameters: variable name, part name, query path, and element index:

```

ora:getElement('variable_name', 'part_name', 'query', index)

```

The previous example using this function would look like this:

```

<assign>
  <copy>
    <from expression="ora:getElement('TicketOffer', 'ticket', '/item',
        bpws:getVariableData('position'))"/>
    <to variable="SelectedOffer" part="ticket"/>
  </copy>
</assign>

```

We usually dynamically address items in loops using the `<while>` activity. To determine the number of items (array size), we can use the Oracle-specific function `ora:countNodes()`. The function returns the number of items as an integer and takes three parameters: variable name, part name, and query path (the last two parameters are optional):

```

ora:countNodes('variable_name', 'part_name', 'query')

```

To count the number of ticket offers in our example we could use the following code:

```

<assign>

```

```

    <copy>
      <from expression="ora:countNodes('TicketOffer',
                                     'ticket',
                                     '/item')"/>
      <to variable="NoOfOffers"/>
    </copy>
  </assign>

```

To append an item to the existing items we can use the Oracle-specific function `ora:addChildNode()`. The syntax of the function is:

```
ora:addChildNode('existing_elements', 'new_item')
```

To add a new ticket offer to the existing offers we can use the following code:

```

<assign>
  <copy>
    <from expression="ora:addChildNode(
                    bpws:getVariableData('TicketOffer', 'ticket'),
                    bpws:getVariableData('NewOffer'))"/>
    <to variable="TicketOffer" part="ticket"/>
  </copy>
</assign>

```

To add more than one item to the existing items, Oracle provides another function called `ora:mergeChildNodes()`. The syntax of the function is:

```
ora:mergeChildNodes('existing_elements', 'new_elements')
```

For example, to add a several new ticket offers to the existing offers we use the following code:

```

<assign>
  <copy>
    <from expression="ora:mergeChildNodes(
                    bpws:getVariableData('TicketOffer', 'ticket'),
                    bpws:getVariableData('AdditionalOffers'))"/>
    <to variable="TicketOffer" part="ticket"/>
  </copy>
</assign>

```

We have seen that Oracle-specific functions simplify array management considerably. Next we look at functions related to XML manipulation.

XML Manipulation

In some cases our BPEL processes will invoke web services that return strings. The content of these strings is XML. This approach is used by some developers, particularly on the .NET platform. Using such web services with BPEL is problematic because no function exists to parse string content to XML. In programming languages such as Java and C# we use XML parser functions or XML serialization (JAXB in Java).

Oracle therefore provides a custom function called `ora:parseEscapedXML()`. The function takes a string as a parameter and returns structured XML data:

```
ora:parseEscapedXML(string)
```

Let us suppose that the Employee web service returns a string instead of XML. We can parse it using the `ora:parseEscapedXML()` function:

```

<!-- Synchronously invoke the Employee Travel Status web service -->
<invoke partnerLink="employeeTravelStatus"
        portType="emp:EmployeeTravelStatusPT"

```

```

        operation="EmployeeTravelStatus"
        inputVariable="EmployeeTravelStatusRequest"
        outputVariable="EmployeeTravelStatusResponseString" />
<assign>
  <copy>
    <from expression="ora:parseEscapedXML(
      bpws:getVariableData('EmployeeTravelStatusResponseString'))"/>
    <to variable="EmployeeTravelStatusResponse" part="employee"/>
  </copy>
</assign>

```

To perform an inverse operation—convert structured XML to a string—we can use the `ora:getContentAsString()` function. It takes structured XML data as a parameter and returns a string:

```
ora:getContentAsString(XMLElement)
```

To set a value of an XML node, Oracle provides the `ora:setNodeValue()` function with the following syntax:

```
ora:setNodeValue('variable_name', 'part', 'query', 'new_node_value')
```

To get a value of an XML node as a string, we can use the `ora:getNodeValue()` function with the following syntax:

```
ora:setNodeValue(node)
```

To get the node value as an integer instead of a string we can use the `ora:integer()` function:

```
ora:integer(node)
```

To add single quotes to a string we can use the `ora:addQuotes()` function:

```
ora:addQuotes(string)
```

Oracle even provides a function to read the content of a file. The function is called `ora:readFile()` and is often used together with the `ora:parseEscapedXML()` function, which converts the file content to structured XML (if the file content is XML). The syntax of the `ora:readFile()` function is:

```
ora:readFile('file_name')
```

Next, we look at the expressions related to date and time.

Date and Time Expressions

Sometimes in our BPEL processes we need the current date and/or time, for example, to timestamp certain data. For this, we can use the Oracle-specific functions:

- `ora:getCurrentDate()`: Get current date
- `ora:getCurrentTime()`: Get current time
- `ora:getCurrentDateTime()`: Get current date and time

Note that all three functions return strings (and not the date or date/time types). All three functions also take an optional parameter that specifies the date/time format. The format is specified according to `java.text.SimpleDateFormat`. For details, refer to Java API documentation at <http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html>).

To format an XML Schema `date` or `dateTime` to a string representation, which is more suitable for output, Oracle provides the `ora:formatDate()` function. The syntax of the function that returns a string is:

```
ora:formatDate('dateTime', 'format')
```

Once again, the format is specified according to `java.text.SimpleDateFormat` format.

Finally, let's look at functions related to process identification.

Process Identification

Oracle provides several functions related to process identification. With these functions we can get process IDs, URLs, and more. These functions are:

- `ora:getProcessId()`: Returns the ID of the current BPEL process
- `ora:getProcessURL()`: Returns the root URL of the current BPEL process
- `ora:getInstanceId()`: Returns the process instance ID
- `ora:getConversationId()`: Returns the conversation ID used in asynchronous conversations
- `ora:getCreator()`: Returns the process instance creator
- `ora:generateGUID()`: Generates a unique GUID (Globally Unique ID)

E-mail and JMS Messaging Support

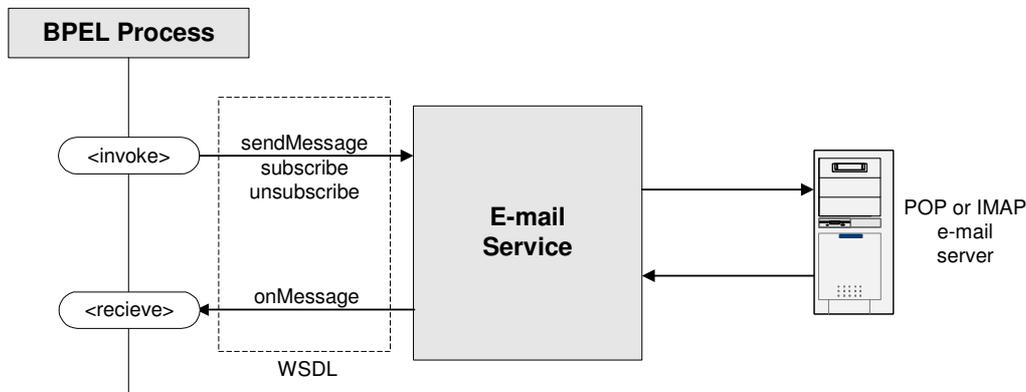
Oracle BPEL Process Manager provides two built-in services to integrate BPEL processes with e-mail and messaging. These services expose their operations like any other web service and are actually wrappers for the underlying e-mail or JMS (Java Message Service) services. So, in order to use them with our own processes we create partner links and then invoke the operations on the corresponding port types. The two built-in services with the WSDL locations are:

- E-mail service: `http://localhost:9700/orabpel/xmllib/MailService.wsdl`
- JMS service: `http://localhost:9700/orabpel/xmllib/JMSService.wsdl`

The Oracle E-mail service offers two port types: `MailService` and `MailServiceCallback`. The `MailService` port type is used to:

- Send e-mail messages (using the `sendMessage` operation)
- Subscribe (or unsubscribe) to be notified about incoming messages (using `subscribe` and `unsubscribe` operations)

The `MailServiceCallback` port type is a callback interface that should be implemented by our BPEL process. It provides the `onMessage` operation through which our process is notified about an incoming e-mail message. All operations require parameters (input messages). Their exact structure will be shown in the next example (can also be seen from the E-mail service WSDL). The following figure shows the architecture of the E-mail service:



The JMS service can be used to integrate BPEL processes with applications using JMS. It is similar to the E-mail service and offers two port types: **JMSservice** and **JMSserviceCallback**. The **JMSservice** port type provides **sendMessage**, **subscribe** and **unsubscribe** operations. The **JMSserviceCallback** interface provides the **onMessage** operation.

E-mail Example

To demonstrate how to use the E-mail service we will add an e-mail confirmation to our travel process example. Originally our process selected the best ticket offer by comparing offers from American and Delta Airlines web services and invoked a callback to the client. We will add an e-mail message confirmation just before the client callback.

Before we start modifying the BPEL code, we need to make modifications to the **TravelRequest** message in the travel process WSDL. We must add the e-mail address to which our process will send the confirmation. Therefore we first define an **EmailType** (in the **Travel.wsdl** file):

```
<types>
  <xs:schema elementFormDefault="qualified"
    targetNamespace="http://packtpub.com/bpel/travel/">
    <xs:complexType name="EmailType">
      <xs:sequence>
        <xs:element name="Address" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:schema>
</types>
```

Next we add a new **email** part to the **TravelRequestMessage**:

```
<message name="TravelRequestMessage">
  <part name="employee" type="emp:EmployeeType" />
  <part name="flightData" type="air:FlightRequestType" />
  <part name="email" type="tns:EmailType" />
</message>
```

Now we are ready to modify the BPEL source code (**Travel.bpel** file). First we have to add the namespace declaration to our process. The E-mail service uses the

<http://services.oracle.com/bpel/mail> namespace, so we add the following line to the `<process>` tag:

```
<process name="BusinessTravelProcess"
  targetNamespace="http://packtpub.com/bpel/travel/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:trv="http://packtpub.com/bpel/travel/"
  xmlns:emp="http://packtpub.com/service/employee/"
  xmlns:aln="http://packtpub.com/service/airline/"
  xmlns:mail="http://services.oracle.com/bpel/mail" >
```

Next we add the partner link according to the partner link type definition in the E-mail service WSDL. Let's call the link `MailService` and the partner role `MailServiceProvider`. The role of our process is `MailServiceRequester`:

```
... <partnerLink name="MailService"
  partnerLinkType="mail:MailService"
  partnerRole="MailServiceProvider"
  myRole="MailServiceRequester"/>
...
```

E-mail service (and JMS service) partner links can be added using BPEL Designer UDDI Browser (select built-in BPEL services).

Sending E-mails

Next we add the e-mail confirmation code. We will send an e-mail confirmation after we have checked that the ticket has been approved and before invoking the callback to the client. We put all code related to the e-mail confirmation in a new scope called `EmailConfirmation`, where we also declare the required variables. We call the variable that holds the e-mail message sent by our process `mailMsg`. The e-mail reply message is stored in the variable `mailResponse`. Both variables are of type `mailEnvelope`. To subscribe the process to the incoming messages we need the `subscriptionRequest` variable:

```
<scope name="EmailConfirmation">
  <variables>
    <variable name="mailMsg"
      messageType="mail:mailEnvelope"/>
    <variable name="subscriptionRequest"
      messageType="mail:subscriptionMessage"/>
    <variable name="mailResponse"
      messageType="mail:mailEnvelope"/>
  </variables>
  ...
</scope>
```

Next we create the content of the send e-mail message and copy it to the `mailMsg` variable. We use the following assign:

```
... <sequence>
  <assign>
    <!-- Create the mail message -->
    <copy>
      <from>
        <mailMessage xmlns="http://services.oracle.com/bpel/mail">
          <from>
```

```

        <email>your.email@address.com</email>
    </from>
    <replyTo>
        <email>your.email@address.com</email>
    </replyTo>
    <to>
        <address>
            <email/>
        </address>
    </to>
    <subject/>
    <mailAccount>TravelEmailAccount</mailAccount>
    <contentType>text/plain</contentType>
    <content/>
</mailMessage>
</from>
<to variable="mailMsg" part="payload"/>
</copy>
...

```

Next we copy the 'to' e-mail address from the **TravelRequest** message (client input) to the 'to' address:

```

...
    <!-- Add the email to address -->
    <copy>
        <from variable="TravelRequest" part="email"
            query="/email/Address"/>
        <to variable="mailMsg" part="payload"
            query="/mailMessage/to/address/email"/>
    </copy>
...

```

We also create the message subject and copy the travel response confirmation XML data into the message body:

```

...
    <!-- Add the message subject -->
    <copy>
        <from expression="concat('Travel confirmation for ',
            bpws:getVariableData('TravelRequest',
                'employee', '/employee/LastName'))"/>
        <to variable="mailMsg" part="payload"
            query="/mailMessage/subject"/>
    </copy>

    <!-- Add the message content -->
    <copy>
        <from variable="TravelResponse" part="confirmationData" />
        <to variable="mailMsg" part="payload"
            query="/mailMessage/content"/>
    </copy>
</assign>
...

```

Now we are ready to send the e-mail message. To do this, we have to invoke the **sendMessage** operation on the **MailService** partner link. Note that Oracle E-mail service is actually a wrapper that provides access to e-mail via web services. So, we can use the service in the same way as any other partner web service:

```

...
    <!-- Send the email by invoking the service -->
    <invoke partnerLink="MailService"
        portType="mail:MailService"

```

```

        operation="sendMessage"
        inputvariable="mailMsg"/>
    ...

```

Receiving E-mail Confirmations

Suppose we want the user to confirm the travel arrangement by replying to the e-mail message before completing the process. To implement this we will subscribe our BPEL process to the e-mail account and then wait for the `onMessage` callback. The E-mail service will invoke the `onMessage` callback once the reply e-mail has been received.

To subscribe to the e-mail service we first have to create the subscription request. We will also add a filter to limit the subscription to the e-mail message with the specified subject and from address:

```

...
<!-- crate the subscription request -->
<assign>
    <copy>
        <from>
            <subscription xmlns="http://services.oracle.com/bpel/mail">
                <mailAccount>TravelEmailAccount</mailAccount>
                <filter/>
            </subscription>
        </from>
        <to variable="subscriptionRequest" part="payload"/>
    </copy>

    <!-- Add a filter by subject and from address -->
    <copy>
        <from expression="concat('subject=&quot;',
            bpws:getVariableData('mailMsg','payload','/mailMessage/subject'),
            '&quot; and from=&quot;',
            bpws:getVariableData('TravelRequest','email','/email/Address'),
            '&quot;')"/>
        <to variable="subscriptionRequest" part="payload"
            query="/subscription/filter"/>
    </copy>
</assign>
...

```

Then we will register our process for the incoming e-mail message by invoking the `subscribe` operation on the `MailService`:

```

...
<!-- Register subscription by invoking the service -->
<invoke partnerLink="MailService"
    portType="mail:MailService"
    operation="subscribe"
    inputvariable="subscriptionRequest"/>
...

```

Finally our process will wait for the callback. Therefore we add a `<receive>` activity for the `onMessage` operation:

```

...
<!-- wait for the confirmation email -->
<receive partnerLink="MailService"
    portType="mail:MailServicecallback"
    operation="onMessage"
    variable="mailResponse"/>

</sequence>
</scope>

```

Configuring an E-mail Account

To make the e-mail example work, we also have to set up an e-mail account we will use. When sending and subscribing to the e-mail we have declared that we will use the **TravelEmailAccount**:

```
<mailAccount>TravelEmailAccount</mailAccount>
```

We create the **TravelEmailAccount.xml** file with the following content:

```
<mailAccount xmlns="http://services.oracle.com/bpel/mail/account">
  <userInfo>
    <displayName>[display name]</displayName>
    <organization>[organization name]</organization>
    <replyTo>[replyTo email address]</replyTo>
  </userInfo>

  <outgoingServer>
    <protocol>smtp</protocol>
    <host>[outgoing smtp server]</host>
    <authenticationRequired>>false</authenticationRequired>
  </outgoingServer>

  <incomingServer>
    <protocol>pop3</protocol>
    <host>[incoming pop3 server]</host>
    <email>[email address]</email>
    <password>[email password]</password>
  </incomingServer>
</mailAccount>
```

Remember to provide details of a valid e-mail account. We then copy this file to the BPEL Server domain. Since we are using the default domain, we copy the file to the following directory:

c:\orabpel\domains\default\metadata\MailService.

We are now ready to compile, deploy, and test the example. The source code can be downloaded from <http://www.packtpub.com/>.

Integration with Java

Sometimes we need to integrate our BPEL processes with resources other than web services. In the Java world this could be EJBs (Enterprise Java Beans), JMS (Java Message Service), ERP systems accessible through JCA (Java Connector Architecture), JDBC databases, or even simple Java classes. Accessing these resources from BPEL processes natively is important because many existing systems use these technologies and we often cannot convert all existing resources to web services before using them in BPEL processes.

Oracle BPEL Process Manager provides native integration with Java. This extends the reach of BPEL and makes it suitable for EAI (Enterprise Application Integration). BPEL Process Manager offers two solutions to integrate Java resources:

- **Java embedding:** This allows us to embed Java code within a BPEL process.
- **Web Services Invocation Framework (WSIF) with Java binding:** This is covered in the next section.

Let's look at Java embedding. Oracle provides a custom BPEL activity called `<exec>`, defined in the <http://schemas.oracle.com/bpel/extension> namespace. This namespace is usually declared with the `bpelx` prefix, so we write the activity as `<bpelx:exec>`.

The `<bpelx:exec>` activity allows us to embed Java code within BPEL processes. The server will execute the embedded Java code within its JTA (Java Transaction API) transaction context. If the embedded Java code calls EJBs (session or entity beans), the transactional context will be automatically propagated. If an exception occurs during the execution of the embedded Java code, the exception will automatically be converted to a BPEL fault and thrown to the BPEL process.

The `<bpelx:exec>` activity supports three attributes (in addition to the BPEL standard attributes):

- **import**: Used to import Java packages.
- **language**: Denotes the used language. Currently the only supported language is Java, but support for other languages such as C# may be added.
- **version**: Denotes the version of the language. The supported version of Java is 1.4.

The `<bpelx:exec>` activity also provides built-in methods we can use in the embedded Java code. They allow us to access and update BPEL variables, get JNDI access, update the audit trail, and set priorities and other parameters. These built-in methods are explained in the following table:

Method	Description
<code>Object getVariableData(String name)</code>	Access BPEL variables
<code>Object getVariableData(String name, String partOrQuery)</code>	
<code>Object getVariableData(String name, String part, String query)</code>	
<code>void setVariableData(String name, Object value)</code>	Update BPEL variables
<code>void setVariableData(String name, String part, Object value)</code>	
<code>void setVariableData(String name, String part, String query, Object value)</code>	
<code>void addAuditTrailEntry(String message, Object detail)</code>	Add an entry or an exception to the audit trail
<code>void addAuditTrailEntry(Throwable t)</code>	
<code>Object lookup(String name)</code>	JNDI lookup
<code>Locator getLocator()</code>	Access to BPEL Process Manager Locator service
<code>long getInstanceId()</code>	Returns the process instance unique ID
<code>void setTitle(String title)</code>	Set/get the title of the process instance
<code>String getTitle()</code>	
<code>void setStatus(String status)</code>	Set/get the status of the process instance
<code>String getStatus()</code>	
<code>void setPriority(int priority)</code>	Set/get the priority of the process instance
<code>int getPriority()</code>	
<code>void setCreator(String creator)</code>	Set/get the creator of the process instance
<code>String getCreator()</code>	


```

<bpelx:exec name="invokeJavaExec" language="java" version="1.4">
  <![CDATA[
    EmployeeStatus e = new EmployeeStatus();
    String firstName = ((Element)getVariableData(
      "EmployeeTravelStatusRequest", "employee",
      "/employee/FirstName")).getNodeValue();
    String lastName = ((Element)getVariableData(
      "EmployeeTravelStatusRequest", "employee",
      "/employee/LastName")).getNodeValue();

    String empStatus = e.getTravelStatus(firstName, lastName);
    addAuditTrailEntry("Employee status is: " + empStatus);
    setVariableData("EmployeeTravelStatusResponse", "travelClass",
      "/travelClass", empStatus);
  ]]>
</bpelx:exec>

```

We have seen that invoking Java resources from BPEL is rather straightforward. For this example to work, we have to pack the Java class file in the BPEL process suitcase JAR archive. We have to store it into the **BPEL-INF/classes** directory. We could invoke an EJB, JMS, JCA, or other Java resources in the same way.

XML Façades and Schema Compiler

Looking at the embedded Java code, we can see that most lines of code have been used to access the BPEL variables and map individual values to Java variables. With more complex variables this can become time consuming and error prone. Instead of hand coding the access to BPEL variables, we can use **XML façades**.

XML façades are a set of Java interfaces and classes through which we can access and modify BPEL (and other XML) variables using get/set methods. The concept is known as **XML serialization** and is also used in JAXB (Java API for XML Bindings). The idea behind XML façades is to generate Java classes from XML Schemas.

Let us demonstrate this with an example. The **EmployeeTravelStatusRequest** variable is defined by the **EmployeeType** complex XML type (located in the **Employee.wsdl** file):

```

<xs:schema elementFormDefault="qualified"
  targetNamespace="http://packtpub.com/service/employee/">
  <xs:complexType name="EmployeeType">
    <xs:sequence>
      <xs:element name="FirstName" type="xs:string" />
      <xs:element name="LastName" type="xs:string" />
      <xs:element name="Departement" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>

```

An XML façade for this variable consists of an interface (**IEmployeeType**) and a class (**EmployeeType**) which provides the following methods:

- **getFirstName()** and **setFirstName()**
- **getLastName()** and **setLastName()**

- `getDepartement()` and `setDepartement()`

There is also a factory class (`EmployeeTypeFactory`) through which we can create the `IEmployeeType` using the `createFacade()` method.

Oracle BPEL Process Manager provides a schema compiler utility called `schemac`. Using this we can generate XML façades. To generate the XML façade for `Employee.wsdl` we can use the following command line:

```
schemac -d ./BPEL-INF/classes Employee.wsdl
```

With the `-d` option we have defined the directory where the generated façade classes should be stored. To see the façade source code we can use the `-trace` option:

```

C:\Prg\orabpel\Chapter4\5_TravelJavaClassFacade>schemac -d ./BPEL-INF/classes Employee.wsdl
-----
Oracle XML Schema Processor Version 2.0
http://otn.oracle.com/bpel
Copyright (c) 2002-2004 - Oracle
(type schemac -help for help)
-----

schemac> parsing schema file 'Employee.wsdl' ...
schemac> Loaded schemas from wsdl located at Employee.wsdl
schemac> generating XML business document ...
schemac> compiling XML business documents ...
Schemac completed successfully.

C:\Prg\orabpel\Chapter4\5_TravelJavaClassFacade>

```

Let us now implement the Java embedded code. First we have to import the XML façade:

```

...
<bpelx:exec import="org.w3c.dom.Element"/>
<bpelx:exec import="com.packtpub.EmployeeStatus"/>
<bpelx:exec import="com.packtpub.service.employee.*"/>
...

```

Then we can modify the code that accesses the BPEL variables. First we have to obtain the DOM element using the `getVariableData()` function. We then create the XML façade and use it to access the first and the last name. Because the façade can throw an exception we have to introduce a try/catch block:

```

...
<!-- Invoke the EmployeeStatus Java class instead of a web service -->
<bpelx:exec name="invokeJavaExec" language="java" version="1.4">
  <![CDATA[
    try {
      EmployeeStatus e = new EmployeeStatus();
      Element empRequest = (Element)getVariableData(
        "EmployeeTravelStatusRequest",
        "employee", "/employee");
      IEmployeeType emp = EmployeeTypeFactory.createFacade(empRequest);
      String firstName = emp.getFirstName();
      String lastName = emp.getLastName();
    }
  ]>

```

```

        String empStatus = e.getTravelStatus(firstName, lastName);
        addAuditTrailEntry("Employee status is: " + empStatus);
        setVariableData("EmployeeTravelStatusResponse", "travelClass",
            "/travelClass", empStatus);
    }
    catch(Exception e)
    {
        addAuditTrailEntry(e);
    }
}]]>
</bpelx:exec>

```

We can see that using the XML façade makes the code simpler and easier to maintain; this is particularly true for larger variables with many member fields. For this example to work, we have to include the XML façade classes in the BPEL process suitcase.

Web Services Invocation Framework Bindings

Integration of Java code into BPEL processes to invoke Java resources is useful. However, such an approach also has disadvantages. In our previous example we had to modify the BPEL process code in order to invoke a Java class instead of the Employee web service. Embedding Java code into BPEL is also a proprietary approach and works only with Oracle BPEL Process Manager.

A much better approach would be if we only needed to modify the service binding and not the BPEL process to replace the Employee web service with a Java class. This is exactly what the WSIF offers. WSIF extends the web services model. It allows us to describe each service in WSDL (even if it is not a web service that communicates through SOAP). It also allows us to map such a service to the actual implementation and protocol.

In other words, we can bind the abstract description of the Employee web service (the port types) to a SOAP-based implementation, to a Java class, to an EJB, or any other supported resource simply by modifying the WSDL binding. No code changes in the BPEL process are necessary. The bindings supported are determined by the providers offered by the WSIF. Oracle BPEL Process Manager currently supports providers for:

- HTTP GET and POST resources
- Java classes
- EJBs
- JCA

In the future support for JMS will be added. Providers support WSDL bindings and allow the invocation of the service through particular implementations.

With WSIF we can integrate resources other than web services into BPEL processes by modifying the WSDL of the services. No changes in the BPEL code are required.

This approach is suitable for real-world scenarios and makes BPEL very useful for EAI as well as for B2B. Enterprise information systems usually consist of a large number of different software

pieces, such as legacy applications accessible through JCA, EJBs, messaging infrastructure (accessible via JMS), web services developed on different platforms, etc. To integrate all these pieces we have to deal with different protocols. If software we use migrates to a different server or has been upgraded to use a new technology, we have to upgrade the integration code—unless we use WSIF. WSIF allows us to describe all these services with WSDL and then bind them to the actual software through providers. It actually separates the interface and the protocol. This gives us the flexibility to change the protocol (and implementation technology) without the need to modify (or even recompile) the BPEL code.

WSIF is an Apache technology that was originally developed by IBM alphaWorks as a part of WSTK (Web Services Toolkit). Oracle has implemented WSIF in the BPEL Process Manager. For more information of WSIF, visit <http://ws.apache.org/wsif/>.

Invoking a Java Class through WSIF

To demonstrate how WSIF works, let's invoke a Java class. Remember that with WSIF we will only have to modify the WSDL of the service and not the BPEL code. So, in this example we will use the original BPEL code that invokes the Employee service using the `<invoke>` activity.

Instead of invoking the Employee web service we will bind it to a Java class. In order to replace the web service with a Java class we will need a class with the exactly the same interface as the web service. We need to modify the Java class from our previous example slightly.

Looking at the Employee web service interface, we can see that it provides an operation that takes as input the `EmployeeTravelStatusRequestMessage`, which is of type `EmployeeType`. To map the `EmployeeType` to Java we use the corresponding XML façade (using the `schemac` tool) as we did in our previous example. The web service operation returns the `EmployeeTravelStatusResponseMessage` message of type `TravelClassType`, which is actually a specialization of `xs:string`. We map this type to `java.lang.String`. We will call the new Java class `EmployeeStatusFull`:

```
package com.packtpub;
import com.packtpub.service.employee.*;
public class EmployeeStatusFull {
    public String getTravelStatus (EmployeeType emp) {
        System.out.println("Java employee status for "+emp.getFirstName()+
            " "+emp.getLastName()+": Economy.");
        return "Economy";
    }
}
```

We added a console output to verify that our process calls the Java class and not the web service.

Next we modify the Employee WSDL. We have to add the binding section, which defines the Java provider to be used. We also have to map the XML types to Java types and the WSDL operation to the Java method. We start by defining the two namespaces used by WSIF providers:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

```

xmlns:tns="http://packtpub.com/service/employee/"
targetNamespace="http://packtpub.com/service/employee/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
xmlns:format="http://schemas.xmlsoap.org/wsdl/formatbinding/"
xmlns:java="http://schemas.xmlsoap.org/wsdl/java/" >
...

```

Next we add the binding section (usually after port type declarations and before partner link types). Here we define a Java binding for the `EmployeeTravelStatusPT` port type. We define the type mapping from XML to Java:

- XML `EmployeeType` is mapped to the `com.packtpub.service.employee.EmployeeType` Java class
- XML `TravelClassType` is mapped to `java.lang.String`

We also have to specify that the WSDL operation `EmployeeTravelStatus` is mapped to the Java method `getTravelStatus()`:

```

...
<!-- Java binding -->
<binding name="JavaBinding" type="tns:EmployeeTravelStatusPT">
  <java:binding/>
  <format:typeMapping encoding="Java" style="Java">
    <format:typeMap typeName="tns:EmployeeType"
      formatType="com.packtpub.service.employee.EmployeeType" />
    <format:typeMap typeName="tns:TravelClassType"
      formatType="java.lang.String" />
  </format:typeMapping>
  <operation name="EmployeeTravelStatus">
    <java:operation methodName="getTravelStatus"/>
    <input/>
    <output/>
  </operation>
</binding>
...

```

Next we have to define the Java port and specify that the Employee service will use the `com.packtpub.EmployeeStatusFull` Java class:

```

...
<service name="Employee">
  <documentation>Employee</documentation>
  <port name="JavaPort" binding="tns:JavaBinding">
    <java:address className="com.packtpub.EmployeeStatusFull"/>
  </port>
</service>

```

The rest of the Employee WSDL (including partner link types) has not been changed. We make no changes to the BPEL process code. Notice that we use the same partner link and invoke the `EmployeeStatusFull` Java class with the usual `<invoke>` activity used for invoking the web service, as shown in the following code excerpt:

```

...
        <!-- Synchronously invoke the Employee Travel Status -->
        <invoke partnerLink="employeeTravelStatus"
              portType="emp:EmployeeTravelStatusPT"
              operation="EmployeeTravelStatus"
              inputVariable="EmployeeTravelStatusRequest"
              outputVariable="EmployeeTravelStatusResponse" />
...

```

To test this example we must first generate the XML façade using the **schemac** utility. Then we have to compile the Java class and deploy it (and the XML façade) to the **c:\orabpel\system\classes** directory. Finally we can compile the BPEL and deploy it. If the BPEL successfully invokes the Java class, the BPEL Process Manager console window will show the following output:



In a similar way we could map the Employee web service to an EJB or other supported resources using the corresponding WSIF provider.

BPEL Server APIs

Until now we have discussed how to develop, deploy, and manage BPEL processes on the Oracle BPEL Process Manager. We have also discussed how to integrate BPEL with Java resources. In complex real-world scenarios we may also need to access the BPEL Server functionalities. For example, we might want to develop our own console through which users could monitor active processes, start new process instances, set the priorities, etc. We might also want to integrate user tasks with BPEL processes.

To realize these requirements, BPEL Server provides access to its functionality through a set of APIs. As the Oracle BPEL Server has been developed in Java, these APIs are packages for use by developers. Using them we can develop our own applications that interact with the server and provide information about the state of the process instances, enable their management, and provide other useful information. Oracle provides Javadoc files to help learn how to use these APIs. The BPEL Console also uses these APIs and the source code is provided (a set of JSPs). Developers can use it to learn how to use the APIs.

The BPEL Process Manager provides the following APIs:

- **com.oracle.services.bpel.task**: Used to interact with the user tasks (discussed later in this chapter).
- **com.oracle.bpel.client**: Provides interfaces and classes for accessing server functionality, such as performing operations on activities and introspecting processes deployed on a server domain.
- **com.oracle.bpel.client.auth**: Used to authenticate against a server domain or for administrative authentication.

- `com.oracle.bpel.client.dispatch`: Used to invoke processes (create process instances) that are deployed on a server domain from Java (for example from JSPs).
- `com.oracle.bpel.client.util`: Contains utility classes for HTML and SQL interaction.
- `com.collaxa.xml`: Provides XML and XPath utility classes. This package might be renamed to a `com.oracle` package.
- `com.collaxa.common.util`: Provides access to BPEL Server performance statistics. This package might also be renamed to a `com.oracle` package.

In the next section we will show how to use some of these APIs to develop user tasks and include user interaction in BPEL processes.

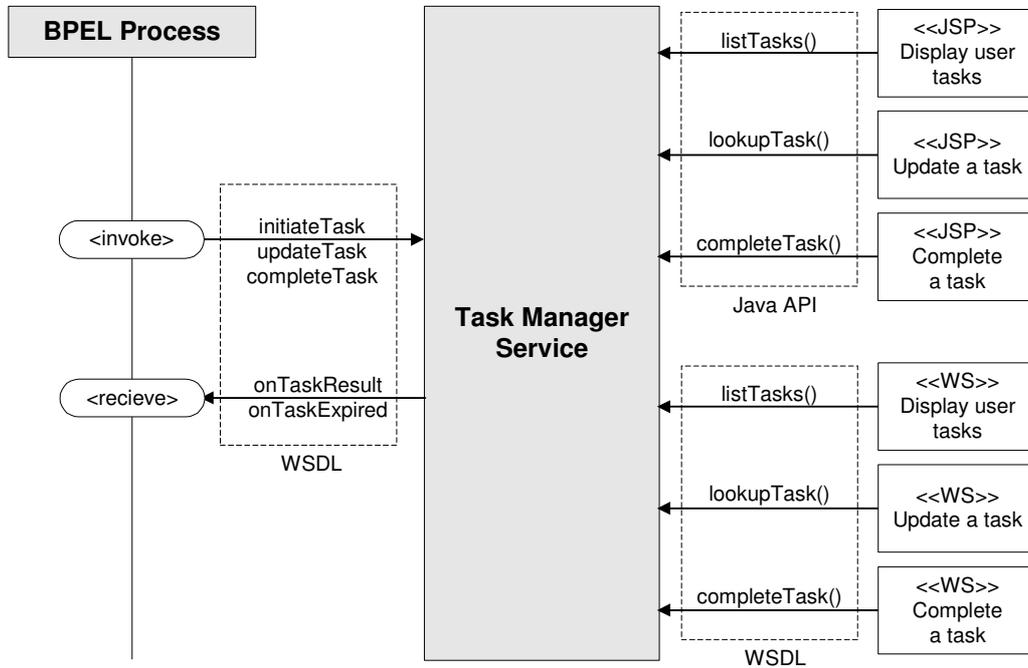
User Interactions and Task Manager

With BPEL we can compose web services (and other resources) into business processes. Real-world business processes sometimes require including user tasks. For example, a user might want to make the final decision about the selected airline ticket, confirm a stock price, or choose a load offer. The BPEL specification does not provide a standard way to include user tasks in BPEL processes.

To solve this problem Oracle BPEL Process Manager provides the **Task Manager**. Task Manager is a built-in BPEL service (similar to E-mail and JMS service), which enables us to include user tasks in BPEL processes. Task Manager is an asynchronous service and provides two interfaces:

- The first is a WSDL interface used by the BPEL process. A BPEL process simply invokes the Task Manager. Through the invocation it expresses the need for the user interaction (`initiateTask` operation). It can also update or complete an existing user task (`updateTask`, `completeTask`). The Task Manager performs a callback to the BPEL process after the user interaction has been completed (`onTaskResult`) or if the user task times out (`onTaskExpired`).
- The second interface of the Task Manager is the client API. Using this API, developers can build custom user interfaces to carry out user interaction. Developers can also list and look up tasks. The client API is available as a Java API (called Worklist API) and can be used to develop user interfaces in Java (JSPs, for example). We will show how to use the Java API in the next example. The client API is also available as a WSDL interface. This enables custom user interfaces to be implemented in Microsoft .NET, Adobe Forms, or any other client technology that supports web services. The client WSDL interface is not available by default and has to be deployed through Worklist Manager service, which is actually a wrapper for the Java Worklist API (`c:\orabpel\samples\utils\worklistManager`).

The architecture of the Task Manager is shown in the following figure:



User Task Example

To demonstrate how to add a user task to a BPEL process, consider our travel process example. We will add a user task to confirm the selected airline ticket. We will proceed as follows:

- Modify the BPEL process to invoke the Task Manager
- Develop a custom user interface using JSPs
- Deploy and test the example

Modifying the BPEL Process

We first declare an additional namespace that is used by the Task Manager:

```

<process name="BusinessTravelProcess"
  targetNamespace="http://packtpub.com/bpel/travel/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:trv="http://packtpub.com/bpel/travel/"
  xmlns:emp="http://packtpub.com/service/employee/"
  xmlns:aln="http://packtpub.com/service/airline/"
  xmlns:task="http://services.oracle.com/bpel/task" >
  
```

We then add a partner link. We will call the partner link `userTaskManager`. The location of the Task Manager WSDL is <http://localhost:9700/orabpel/default/TaskManager/TaskManager?wsdl>:

```

... <partnerLink name="userTaskManager"
  
```

```

partnerLinkType="task:TaskManager"
partnerRole="TaskManager"
myRole="TaskManagerRequester" />

```

...

Next we invoke the Task Manager. For this we create a new scope just after the `<switch>` activity, where we select the best plane ticket offer. To initiate a user task we invoke the `initiateTask` operation on the Task Manager. This operation requires a `taskMessage` parameter that specifies the task details. Therefore we create the `taskMessage`, fill in the required data, and invoke the `initiateTask` operation. Finally we wait for the callback `onTaskResult` (when the user has approved the ticket) and copy the user input to the `TravelResponse` variable:

```

<!-- User task to approve the ticket -->
<scope name="ApproveTicket">
  <variables>
    <variable name="ApproveTask" element="task:task"/>
  </variables>

  <sequence>
    <assign>
      <!-- Assign 'title' in task document -->
      <copy>
        <from expression="string('Approve Ticket')"/>
        <to variable="ApproveTask" query="/task/title"/>
      </copy>
      <!-- Assign 'creator' in task document -->
      <copy>
        <from expression="string('TravelUserTask')"/>
        <to variable="ApproveTask" query="/task/creator"/>
      </copy>
      <!-- Assign 'assignee' in task document -->
      <copy>
        <from expression="string('travel@packtpub.com')"/>
        <to variable="ApproveTask" query="/task/assignee"/>
      </copy>
      <!-- Assign 'duration' in task document -->
      <copy>
        <from expression="string('PT1H')"/>
        <to variable="ApproveTask" query="/task/duration"/>
      </copy>
      <!-- Assign 'priority' in task document -->
      <copy>
        <from expression="3"/>
        <to variable="ApproveTask" query="/task/priority"/>
      </copy>
      <!-- Assign 'attachment' in task document -->
      <copy>
        <from variable="TravelResponse" part="confirmationData"/>
        </from>
        <to variable="ApproveTask" query="/task/attachment"/>
      </copy>
    </assign>

    <scope name="UserInteraction">
      <variables>
        <variable name="taskRequest" messageType="task:taskMessage"/>
        <variable name="taskResponse" messageType="task:taskMessage"/>
      </variables>

      <sequence>
        <!-- Assign task document to taskMessage -->
        <assign>
          <copy>
            <from variable="ApproveTask"/>

```

```

        <to variable="taskRequest" part="payload"/>
    </copy>
</assign>

<!-- Initiate task -->
<invoke partnerLink="userTaskManager"
    portType="task:TaskManager"
    operation="initiateTask"
    inputVariable="taskRequest" />

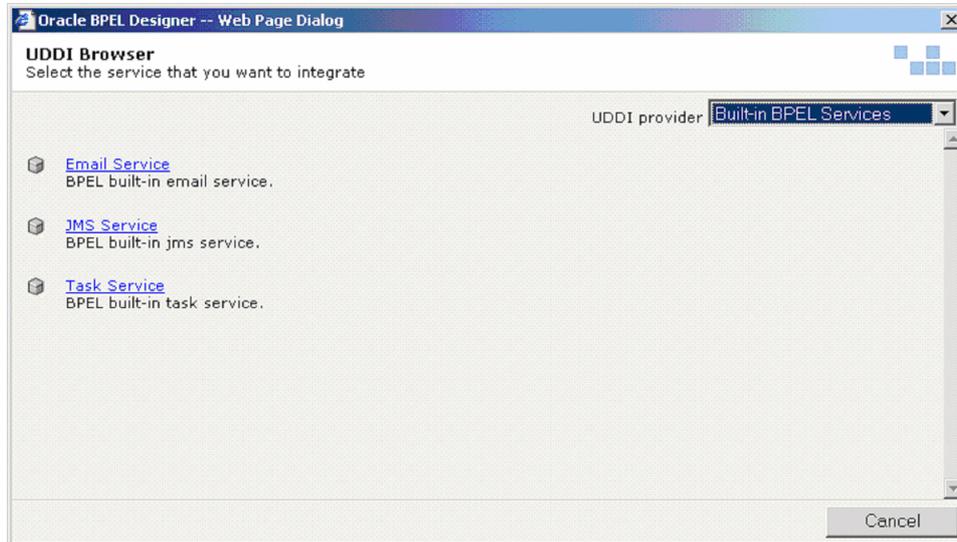
<!-- Receive the outcome of the task -->
<receive partnerLink="userTaskManager"
    portType="task:TaskManagerCallback"
    operation="onTaskResult"
    variable="taskResponse" />

<!-- Read task document from taskMessage -->
<assign>
    <copy>
        <from variable="taskResponse" part="payload"/>
        <to variable="ApproveTask"/>
    </copy>
</assign>
</sequence>
</scope>
<!-- Copy updated task attachment to variable -->
<assign>
    <copy>
        <from variable="ApproveTask" query="/task/attachment"/>
        <to variable="TravelResponse" part="confirmationData"/>
    </copy>
</assign>
</sequence>
</scope>

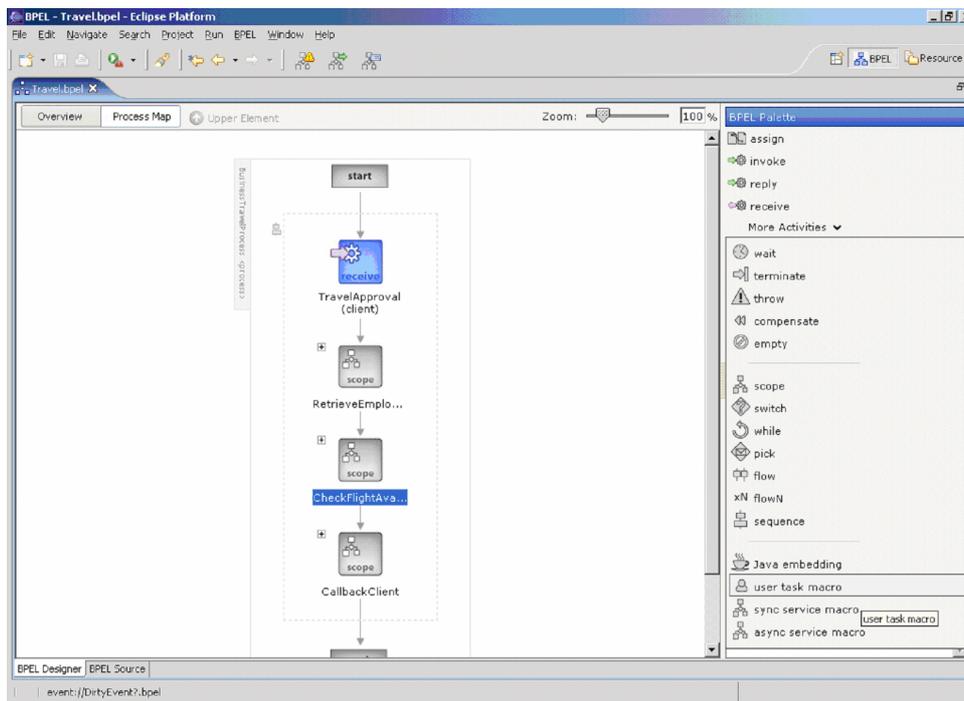
```

Using BPEL Designer to Add a User Task

Instead of adding the Task Manager partner link and the related code by hand we could use the BPEL Designer. To add the partner link we can use the UDDI Browser and select the built-in BPEL Services, as shown in the following screenshot:



To add the code for invoking the Task Manager we can use the User Task macro and drag-and-drop it to the process, as shown:



We will have to enter the task name and finally modify the assignments to add the task data. For more information about using BPEL Designer, refer to Oracle documentation.

Developing the Custom User Interface

After we have successfully modified the BPEL code we are ready to develop the custom user interface through which the user will approve the airline tickets. We will develop three JSPs:

- One to display the tasks waiting for approval
- One to display and enter the airline ticket information
- One to make the ticket confirmation

To simplify the data management we will use the XML façade for the **TravelResponse** message. For this we use the **schemac** tool on the Airline WSDL. Let us now develop the first JSP.

Displaying Tasks

To display the tasks that are waiting for approval, we use the BPEL Server Locator through which we connect to the BPEL default domain. Next we connect to the Worklist service and get the list of tasks that require approval:

```
<%@page import="java.util.Date" %>
<%@page import="com.oracle.bpel.client.Locator" %>
<%@page import="com.oracle.services.bpel.task.IworklistService" %>
<%@page import="com.oracle.services.bpel.task.ITask" %>

<%
    // This page should not be cached
    response.setHeader("Pragma", "no-cache");
    response.setHeader("Cache-Control", "no-cache");

    // Connect to the default BPEL domain using Locator
    // Please set the password (bpel is initial password)
    Locator locator = new Locator("default", "bpel");

    // Look up the worklist service
    IworklistService worklist =
        (IworklistService)locator.lookupService(IworklistService.SERVICE_NAME);

    // List of tasks assigned to confirm
    ITask[] tasks = worklist.listTasksByAssignee("travel@packtpub.com");
%>
...

```

Next we list the tasks in the table and make a hyperlink to the next JSP through which we display the ticket information. Note that we use the task ID to identify which task the user has selected:

```
...
<html>
  <head>
    <meta http-equiv="PRAGMA" content="NO-CACHE" />
    <meta http-equiv="EXPIRES" content="-1" />
  </head>
  <body>
    <h1>BPEL Travel process tasks waiting for approval</h1>
    <table>
      <%
        for ( int i = 0; i < tasks.length; i ++ )
        {
          ITask thisTask = tasks[i];

```

```

// select only tasks that belong to us
if ( ! "TravelUserTask".equals( thisTask.getCreator() ) )
{
    continue;
}

// Get the task title
String title = thisTask.getTitle();

// Get the task ID
String taskId = thisTask.getTaskId();

// Get the task expiration date
Date expiration = null;
if( thisTask.getExpirationDate() != null )
    expiration = thisTask.getExpirationDate().getTime();

%>
|  |  |
| --- | --- |
| <%= expiration %> | <a href="displayTicket.jsp?taskId=<%= taskId %>"><%= title %></a> |

<%
}
%>
</table>

</body>
</html>

```

Displaying and Entering Ticket Information

In the second JSP we display information about the airline ticket and allow the user to edit the approval field (which can be true or false). We again use the Locator to connect to the BPEL domain. Then we connect to the Worklist service and locate the task by ID. We use the XML façade to obtain the ticket data:

```

<%@ page import="java.util.*" %>
<%@ page import="org.w3c.dom.Element" %>
<%@ page import="com.oracle.bpel.client.Locator" %>
<%@ page import="com.oracle.services.bpel.task.IWorklistService" %>
<%@ page import="com.oracle.services.bpel.task.ITask" %>
<%@ page import="com.packtpub.service.airline.FlightConfirmationType" %>
<%@ page import="com.packtpub.service.airline.FlightConfirmationTypeFactory" %>
%>

<html>
<head>
<meta http-equiv="PRAGMA" content="NO-CACHE" />
<meta http-equiv="EXPIRES" content="-1" />
</head>
<body>
<%
String taskId = request.getParameter( "taskId" );
if( taskId == null || "".equals( taskId ) )
{
%>
<a href="displayTasks.jsp">
    List of BPEL Travel process tasks to approve.</a>
<%
}
else

```

```

{
    // Connect to the default BPEL domain using Locator
    // Please set the password (bpe1 is initial password)
    Locator locator = new Locator( "default", "bpe1" );

    // Lookup the worklist service
    IworklistService worklist =
        (IworklistService)locator.lookupService(
            IworklistService.SERVICE_NAME );

    // Lookup the specific task the user has to confirm
    ITask task = worklist.lookupTask( taskId );

    // Get the data using XML facade
    Element rsElement = (Element) task.getAttachment();
    FlightConfirmationType fc =
        FlightConfirmationTypeFactory.createFacade(rsElement);
    String flightNo = fc.getFlightNo();
    String travelClass = fc.getTravelClass();
    float price = fc.getPrice();
    boolean approved = fc.getApproved();
}
...
%>

```

We display the ticket data in a form and allow the user to edit the approved field. The form is linked to the third JSP:

```

... <h1>Travel Ticket Approval User Task</h1>
<form action="confirmTicket.jsp" method="POST">
    <!-- The task ID is passed from page to page -->
    <input type="hidden" name="taskId" value="<%=taskId%>" />
    <table>
        <tr>
            <td>Flight number</td>
            <td><%= flightNo %></td>
        </tr>
        <tr>
            <td>Travel class</td>
            <td><%= travelClass %></td>
        </tr>
        <tr>
            <td>Price</td>
            <td><%= price %></td>
        </tr>
        <tr>
            <td>Approved</td>
            <td>
                <input type="text" name="approved" value="<%= approved %>" />
            </td>
        </tr>
    </table>
    <input type="submit" alt="Confirm Ticket" value="Confirm Ticket"/>
</form>
<%
}
%>
</body>
</html>

```

Ticket Confirmation

In the third JSP we set the user input regarding the ticket approval. We again use the XML façade. Then we notify the Task Manager that the user task is completed. The Task Manager will invoke the callback to the BPEL process:

```
<%@ page import="java.util.*" %>
<%@ page import="org.w3c.dom.Element" %>
<%@ page import="com.oracle.bpel.client.Locator" %>
<%@ page import="com.oracle.services.bpel.task.ITask" %>
<%@ page import="com.oracle.services.bpel.task.IworklistService" %>
<%@ page import="com.packtpub.service.airline.FlightConfirmationType" %>
<%@ page import="com.packtpub.service.airline.FlightConfirmationTypeFactory"
%>

<html>
<head>
<meta http-equiv="PRAGMA" content="NO-CACHE" />
<meta http-equiv="EXPIRES" content="-1" />
</head>
<body>
<h1>Travel Ticket Approval User Task</h1>
<%
String taskId = request.getParameter( "taskId" );
if( taskId == null || "".equals( taskId ) )
{
%>
<a href="displayTasks.jsp">List of BPEL Travel process tasks to
approve.</a>
<%
}
else
{
// Connect to the default BPEL domain using Locator
// Please set the password (bpel is initial password)
Locator locator = new Locator( "default", "bpel" );

// Lookup the worklist service
IworklistService worklist =
(IworklistService)locator.lookupService(IworklistService.SERVICE_NAME);

// Lookup the specific task the user has selected
ITask task = worklist.lookupTask( taskId );

// Set the approved field using XML facade
Element rsElement = (Element) task.getAttachment();
FlightConfirmationType fc =
FlightConfirmationTypeFactory.createFacade(rsElement);
if ((request.getParameter("approved")).equalsIgnoreCase("true")) {
fc.setApproved(true);
} else {
fc.setApproved(false);
}

// Update the attachment so that it reflects the user changes
task.setAttachment(fc.getRootElement());

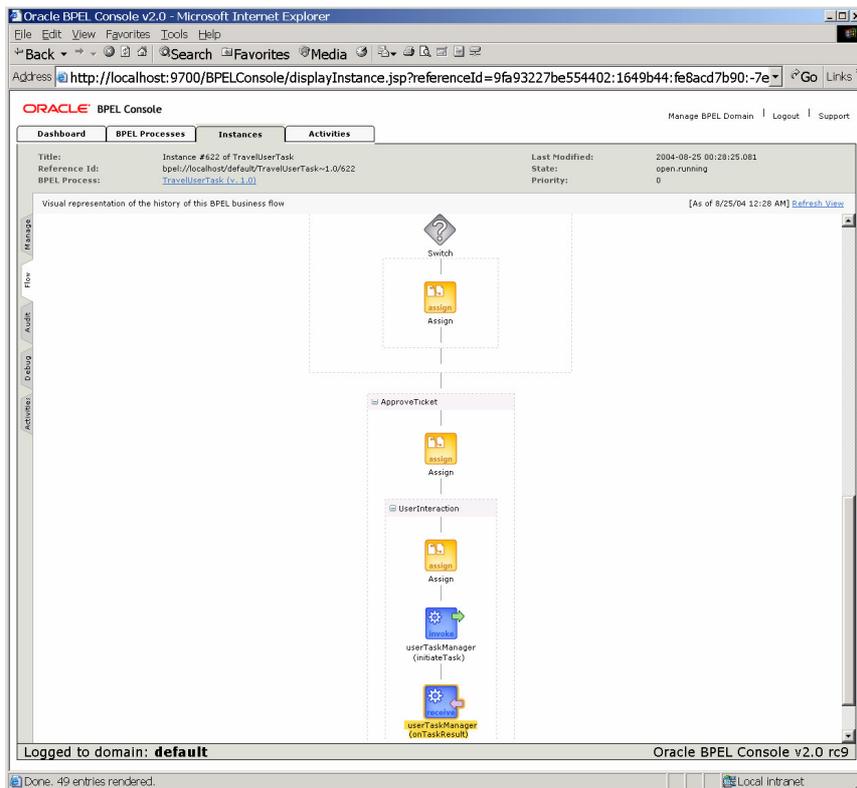
// Complete the task to activate the callback
worklist.completeTask(task);

out.println("Ticket approved status: "+fc.getApproved()+".");
}
%>
</body>
</html>
```

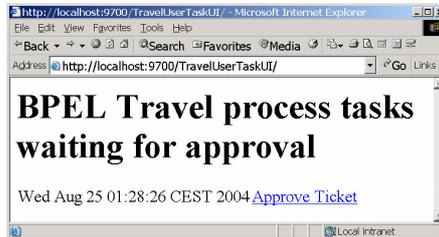
Deployment and Testing

To deploy and test our example, we first have to compile and deploy the BPEL process. We then create a Java WAR web archive and deploy it to the OC4J application server. This can be done using **obant**. For more details look at the example code, which can be downloaded from <http://www.packtpub.com/>.

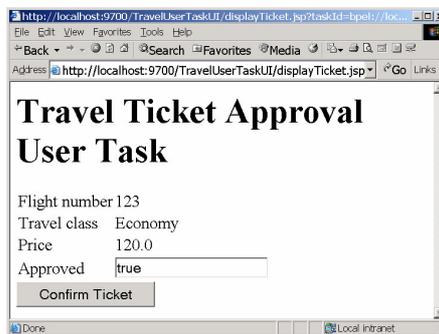
After we have successfully deployed the example we can test it. We use the BPEL Console to initiate the process. From the visual flow we can see that the process has not completed but is waiting for the Task Manager callback:



Now we have to use our custom user interface to approve the ticket. The user interface can be accessed at <http://localhost:9700/TravelUserTaskUI/>:



After we click on the [Approve Ticket](#) link we will see the following screen:



Now we can enter a value in the [Approved](#) field. After clicking on the [Confirm Ticket](#) button we have finished the user task. In the BPEL Console we can observe that the process has now either completed successfully if we have approved the ticket or an exception has been thrown if we have not approved the ticket. We can see that user tasks can be a very useful way to integrate user actions into BPEL processes.

Summary

In this chapter we provided a detailed overview of the Oracle BPEL Process Manager. We saw that Oracle BPEL Process Manager is a J2EE based BPEL server that also provides an integrated graphical development environment called BPEL Designer (an Eclipse plug-in) and a BPEL Console, which can be used for process deployment, monitoring, debugging, and administration. The Oracle BPEL Server provides several advanced features such as dehydration, version control, and clustering.

The Oracle BPEL Process Manager also provides several integration capabilities. It has built-in XSLT, XQuery, and XSQL engines that we can use in our BPEL processes. It supports the Web Services Invocation Framework through which we can include resources other than web services into our BPEL processes by simply specifying the service bindings. Oracle BPEL Process Manager also supports integration with email and messaging. These features extend the usability of BPEL considerably.

Oracle BPEL Process Manager also provides integration with Java. We can embed Java code in BPEL and therefore integrate BPEL processes with Java and J2EE resources (such as EJBs, JCA,

JMS, etc.). We can also access the functionality of the BPEL Process Manager from Java through a set of APIs. In this way we can develop our own consoles and other applications. We can also integrate user tasks with BPEL processes. In this way users can confirm process activities or provide other input to BPEL processes.

Oracle BPEL Process Manager offers a comprehensive, powerful, and relatively easy-to-use environment for the development and deployment of BPEL processes.



Business Process Execution Language for Web Services

Business Process Execution Language for Web Services (BPEL4WS) is the new standard for orchestrating business process using web services. BPEL is supported by more platform vendors than its predecessors that tried to achieve similar goals, such as ebXML and Web Services Choreography Interface (WSCI). BPEL is supported by Microsoft, IBM, BEA, SAP, Hewlett-Packard, Oracle, Siebel, and others.

The book explains the BPEL standard and how it relates to the web services stack and to previous similar standards. It also covers the Microsoft BPEL server—BizTalk, and the Oracle BPEL Process Manager. We will see how these servers use web services and XML for document exchange. The book presents the service oriented architecture for web services development which enable us to develop loosely-coupled solutions.

What This Book Covers

Chapter 1 provides a detailed introduction to Service Oriented Architecture (SOA) and the distributed SOA model. The chapter goes on to discuss the important standards and specifications for implementing SOA with web services and integrating web services.

Chapter 2 discusses the composition of web services with BPEL. The chapter introduces the core concepts of BPEL and explains how to describe synchronous and asynchronous business processes with BPEL. The chapter finishes with an overview of Orchestration Servers.

Chapter 3 goes deeper into the BPEL specification and covers advanced functionality for modeling complex business processes. Advanced activities, scopes, serialization, fault and event handling, and correlation sets are covered in detail.

Chapter 4 explains how to use the Oracle BPEL Process Manager for deploying and running business process defined in BPEL. The chapter also looks at graphical development of BPEL business processes using Oracle BPEL Designer. Important topics such as integrating web services with Java, special Oracle-specific functions, and adding user interactivity (using Oracle Task Manager) are discussed in detail.

Chapter 5 discusses MS BizTalk Server 2004, an integration server product that allows us to import and export business processes to BPEL. The chapter also explains how to use the Orchestration Designer tool to define business processes graphically.

Appendix A provides a syntax reference for BPEL v 1.1. The appendix covers standard BPEL activities and elements, functions, attributes, and faults.

For more information: www.PacktPub.com/book/BPEL

Where to buy this book

You can buy Business Process Execution Language for Web Services direct from the Packt Publishing website: www.PacktPub.com/book/BPEL. This book carries at least a 10% discount on the website as well as free shipping to the US, UK, Europe, Australia & New Zealand.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com